Actes de la conférence JFLA2008 (Journées Francophones des Langages Applicatifs)

Sandrine Blazy

ACTES DE LA CONFÉRENCE

JFLA2008

Journées Francophones des Langages Applicatifs

Responsable scientifique : Sandrine Blazy

26 au 29 janvier 2008
Étretat, France

INSTITUT NATIONAL
DE RECHERCHE
EN INFORMATIQUE
ET EN AUTOMATIQUE

INRIA

Avant-propos

Sandrine Blazy[1] & Alan Schmitt[2]

1: CEDRIC, ENSIIE
2: INRIA Grenoble - Rhône Alpes

JFLA 2008 est la dix-neuvième conférence francophone organisée autour des langages applicatifs. Cette année, les journées ont lieu à Étretat, en Normandie. Nous respectons ainsi l'alternance mer-montagne traditionnelle aux JFLA et le choix d'un cadre propice aux échanges conviviaux sur les langages applicatifs.

Le comité de programme a choisi 11 articles sur les 18 présentés, portant sur des sujets très variés, témoignant de la vitalité et de la diversité de notre communauté. Les soumissions étaient de grande qualité et nous remercions les auteurs qui ont soumis un article aux JFLA 2008, en contribuant ainsi au haut niveau scientifique de ces journées.

Deux orateurs ont été invités : Pierre Weis, de l'INRIA Paris-Rocquencourt et Cédric Fournet, de Microsoft Research. Pour la troisième année consécutive, deux sessions d'une demi-journée chacune ont été consacrées à la découverte de thèmes de recherche : les sémantiques formelles (par Yves Bertot de l'INRIA Sophia Antipolis - Méditerranée) et la conception d'une bibliothèque formelle de mathématiques effectives (par Renaud Rioboo du laboratoire CEDRIC de l'ENSIIE).

Nous remercions chaleureusement les membres du comité de programme : Horatiu Cirstea, Tom Hischowitz, Mathieu Jaume, Delia Kesner, Nicolas Magaud, Marc Pouzet, Laurence Rideau et Francesco Zappa Nardelli.

Nous remercions également Oana Andrei, Paul Brauner, David Delahaye, Jacques Guarrigue, Thérèse Hardin, Clément Houtmann, Pierre Hyvernat et Daniele Varaca qui ont participé à la relecture des articles soumis.

Les JFLA existent grâce au support constant de l'INRIA et nous voulons chaleureusement remercier Gaëlle Dorkeld du Bureau des Cours et Colloques de l'INRIA pour la prise en charge de l'organisation matérielle de ces Journées. Les JLFA ne seraient pas sans l'investissement de Pierre Weis, qui gère le site des JFLA et coordonne donc de main de maître toutes les étapes de préparation des Journées. Qu'il en soit ici très profondément remercié !

Table des matières

Formalisation des mathématiques : une preuve du théorème de Cayley-Hamilton *

Sidi Ould Biha [1]

1: Inria de Sophia-Antipolis,
2004, route des Lucioles - B.P. 93 06902 Sophia Antipolis Cedex, France
Sidi.Ould_biha@sophia.inria.fr
*: *Ce travail a été possible grâce au financement du laboratoire commun Microsoft-Inria*
http://www.msr-inria.inria.fr

Résumé

Le théorème de Cayley-Hamilton est l'un des principaux théorèmes de l'algèbre linéaire. Dans cet article, nous présentons une première formalisation dans un assistant à la preuve de ce théorème. Cette formalisation a été développée dans Coq en utilisant son extension SSReflect développée par G. Gonthier. Ce travail repose sur des développements sur les matrices, les polynômes et les opérations indexées. Il rentre dans le cadre des travaux de formalisation du théorème de Feit-Thompson sur les groupes solvables.

1. Introduction

Les systèmes de preuves formelles peuvent être d'une grande utilité dans la vérification et la validation de preuves mathématiques, surtout lorsque ces preuves sont complexes et longues. Les travaux récents, comme la preuve formelle du théorème des 4 couleurs [6] ou celle du théorème des nombres premiers [2], montrent que ces systèmes ont atteint un niveau de maturité leur permettant de s'attaquer à des problèmes mathématiques non triviaux. Le travail de formalisation de preuves mathématiques faisant intervenir une large variété d'objets mathématiques nécessite l'adoption d'une approche semblable au génie logiciel. La formalisation de telles théories peut être vue comme un développement faisant intervenir différentes composantes : définitions et preuves mathématiques.

Une liste des 100 plus grands théorèmes mathématiques [1] a été constituée par Paul et Jack Abad. Cette liste prend en compte la place du théorème dans la littérature mathématique, la qualité de sa preuve et l'importance du résultat qu'il introduit. F. Wiedijk maintient une liste [13] qui recense les formalisations de ces théorèmes dans différents systèmes de preuves formelles. Le théorème de Cayley-Hamilton est l'un des théorèmes présents dans cette liste. Ce papier présente à notre connaissance, la première formalisation de ce théorème. Le fait qu'il n'avait pas été jusqu'à ce jour formalisé peut s'expliquer par le fait qu'il fait intervenir de nombreux objets et propriétés mathématiques de nature différente (algèbre linéaire, multilinéaire, combinatoire, ..). Ces objets ne sont pas uniquement utilisés de façon indépendante, au contraire ils s'emboîtent les uns avec les autres. Ce travail de formalisation du théorème de Cayley-Hamilton est utilisé dans le cadre des travaux de formalisation du théorème de Feit-Thompson sur les groupes d'ordre impair. L'objectif n'est pas seulement de formaliser Cayley-Hamilton mais d'organiser la preuve en bibliothèques réutilisables.

L'article est organisé comme suit. Dans la section 2, nous présentons l'énoncé et la preuve du théorème de Cayley-Hamilton. Dans la section 3, nous présentons brièvement SSReflect, l'extension

de Coq développée par G. Gonthier pour la preuve du théorème des 4 couleurs, et plate-forme de notre développement. Enfin, dans la section 4, nous présentons le développement qui a été nécessaire pour arriver à la formalisation du théorème de Cayley-Hamilton.

2. Le théorème de Cayley-Hamilton

Le théorème de Cayley-Hamilton, qui porte le nom des mathématiciens Arthur Cayley(1821-1895) et William Hamilton(1805-1865), est un résultat important de l'algèbre linéaire. Il peut être énoncé de la façon suivante :

Toute matrice carrée sur un anneau commutatif annule son polynôme caractéristique.

Plus formellement, soient R un anneau commutatif et A une matrice carrée sur R. Alors, le polynôme caractéristique de A, défini par : $p_A(x) = \det(xI_n - A)$, s'annule en A.
Le théorème peut être énoncé différemment en considérant les endomorphismes d'espace vectoriel. Dans ce cas il n'est plus question d'anneau commutatif mais de corps.

Le théorème de Cayley-Hamilton est utilisé pour faire des calculs sur les matrices carrées (ou les endomorphismes) : calcul de la matrice inverse ou calcul des valeurs propres. Un corolaire de ce théorème est le résultat selon lequel, le polynôme minimal d'une matrice donnée est un diviseur de son polynôme caractéristique.

La preuve du théorème de Cayley-Hamilton présentée dans [4] découle de la formule de Cramer. En notant ${}^t\mathrm{com}B$ la transposée de la co-matrice de B, la règle de Cramer est :

$$B * {}^t\mathrm{com}B = {}^t\mathrm{com}B * B = \det B * I_n \tag{1}$$

En appliquant la formule (1) à la matrice $(xI_n - A) \in M_n(R[x])$, nous obtenons :

$${}^t\mathrm{com}(xI_n - A) * (xI_n - A) = \det(xI_n - A) * I_n = p_A(x) * I_n \tag{2}$$

L'anneau $M_n(R[x])$ des matrices de polynômes est aussi celui des polynômes à coefficients matriciels $(M_n(R))[X]$. L'égalité (2) s'écrit ainsi dans $(M_n(R))[X]$:

$${}^t\mathrm{com}(xI_n - A) * (X - A) = p_A(X) \tag{3}$$

Ceci montre que $(X - A)$ est facteur de $p_A(X)$ dans $(M_n(R))[X]$ et donc $p_A(A) = O_n$.

Formaliser une preuve mathématique dans un assistant à la preuve consiste à développer cette preuve pour qu'elle soit compréhensible pour un ordinateur. Pour arriver à cet objectif deux difficultés sont à surmonter. En premier lieu, il faut expliciter les parties de la preuve qui sont implicites ou "triviales" pour un mathématicien. Paradoxalement, l'implémentation sur ordinateur de ces parties, qui n'apparaissent pas dans la preuve, est la tâche la plus complexe du travail de formalisation. En second lieu, il faut avoir des énoncés compréhensibles pour un mathématicien. L'intérêt n'est pas simplement de faire des preuves sur ordinateurs mais il faut que les énoncés de ces preuves soient le plus proche possible de ceux utilisés dans la littérature mathématique. Ceci facilitera la réutilisation de ces preuves dans d'autres développements.

Dans le cas du théorème de Cayley-Hamilton et en considérant la preuve ci-dessus plusieurs problèmes se posent lors de sa formalisation. Dire que $M_n(R[x])$ est identique à $(M_n(R))[x]$ équivaut algébriquement à dire qu'il existe un isomorphisme d'anneau entre eux. En effet, toute matrice de polynômes peut s'écrire, de façon unique comme la somme de puissances en x multipliées par des matrices, c'est-à-dire un polynôme à coefficients matriciels. Par exemple :

$$\begin{pmatrix} x^2+1 & x-2 \\ -x+3 & 2x-4 \end{pmatrix} = x^2 \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix} + x \begin{pmatrix} 0 & 1 \\ -1 & 2 \end{pmatrix} + \begin{pmatrix} 1 & -2 \\ 3 & -4 \end{pmatrix} \quad (4)$$

La formalisation de cette isomorphisme correspond à l'écriture de la fonction de transformation décrite dans l'exemple ci-dessus. Les propriétés de ce morphisme que nous noterons ϕ sont utilisées implicitement dans la preuve. En effet, dans (2) les membres de l'égalité sont des matrices de polynômes. L'application de ϕ aux parties gauche et droite de (2) nous donne :

$$\phi({}^t\mathrm{com}(xI_n - A) * (xI_n - A)) = \phi(p_A(x) * I_n)$$

Les propriétés de morphisme de ϕ sont alors utilisées pour obtenir :

$$\phi({}^t\mathrm{com}(xI_n - A)) * \phi(xI_n - A) = \phi(p_A(x) * I_n)$$

La formule (3) correspond explicitement à l'égalité ci-dessus.

3. SSReflect

SSReflect [7, 8] (pour *Small Scale Reflection* ou réflection à petite échelle) est une extension de Coq [3] qui introduit de nouvelles tactiques et des bibliothèques Coq adaptées pour travailler sur des types avec une égalité décidable et équivalente à l'égalité structurelle de Coq (égalité de Leibniz). Cette extension a été initialement développée par G. Gonthier dans le cadre de sa preuve du théorème des 4 couleurs. Un développement [8] sur la théorie des groupes finis a été fait au dessus de SSReflect. Il comprend, entre autres, une formalisation du théorème de Sylow et du lemme de Cauchy-Frobenuis.
Dans cette section, nous présentons en premier lieu la méthode de SSReflect pour réfléchir entre les prédicats booléens dans la logique de Coq. Nous introduisons par la suite le langage de tactique de SSReflect. Enfin, nous présentons quelques bibliothèques Coq de SSReflect que nous avons utilisées dans notre développement. Des informations plus détaillées sur SSReflect et précisément son langage de tactique peuvent être obtenues dans [7].

Réflection

Dans le système de preuve Coq la logique par défaut est intuitionniste. Dans cette logique, les propositions logiques et les valeurs booléennes sont distinctes. SSReflect permet de combiner le meilleur des deux visions et passer de la version propositionnelle d'un prédicat décidable vers la version booléenne. La version propositionnelle permet d'avoir des preuves structurées alors que la version booléenne permet de faire des calculs. Pour ce faire, le type booléen est injecté dans celui des propositions par une coercion :

```
Coercion is_true (b: bool) := b = true.
```

Ainsi, et de façon transparente pour l'utilisateur, lorsque Coq attend un objet de type `Prop` et reçoit une valeur `b` de type `bool`, il la traduira automatiquement en la proposition `(is_true b)`, qui correspond à la proposition `b = true`.
Le prédicat inductif `reflect` donne une équivalence pratique et confortable entre les propositions décidables et les booléens :

```
Inductive reflect (P: Prop): bool -> Type :=
  | Reflect_true: P => reflect P true
  | Reflect_false: ~P => reflect P false.
```

La proposition `(reflect P b)` indique que `P` est équivalent à `(is_true b)`. Par exemple, l'équivalence entre la conjonction booléenne `&&` et celle propositionnelle `/\` est donnée par le lemme suivant :

```
Lemma andP: forall a b:bool, reflect (a /\ b) (a && b).
```

Des lemmes de même nature que `andP` sont définis lorsque nous voulons avoir l'équivalence entre la représentation calculatoire d'une fonction (pouvant être calculée) donnée et sa représentation logique. Plus de détails sur l'utilisation de `reflect` sont disponibles dans la documentation de SSReflect [7].

Langage de tactiques

Les scripts de preuve écrits avec SSReflect diffèrent de ceux écrits dans Coq standard. Le langage de tactique de SSReflect permet de faciliter les opérations d'interprétation et le développement des scripts. En pratique, les scripts de preuve écrits avec SSReflect se révèlent plus concis que ceux écrits dans Coq standard.
Toutes les opérations fréquentes qui consistent à déplacer ou généraliser depuis ou vers le contexte courant des formules sont regroupées dans la tactique `move`. Par exemple la tactique "`move: (H1 a)`" permet de placer dans le but courant une instance de l'hypothèse `H1` pour la variable `a`. Un autre exemple est la tactique "`move=> x y H2`" qui correspond à l'introduction des variables `x` et `y`, et d'une nouvelle hypothèse `H2` dans le contexte courant. La tactique `move: (H1 a) => H2 x y` correspond à la combinaison des deux exemples précédents dans une seule et unique tactique.
La tactique `rewrite` permet de combiner toutes les opérations de réécriture conditionnelle, de dépliage de définition, de simplification et de réécriture pour une occurrence ou un pattern donné. Ces opérations peuvent être utilisées ensemble ou séparément. Par exemple la tactique `rewrite /def H1 ?H2 !H3 {2}[_ * _]H4 /=` permet de déplier la définition `def`, de récrire avec l'hypothèse `H1`, de récrire zéro ou plusieurs fois avec l'hypothèse `H2`, de récrire au moins une fois avec l'hypothèse `H3`, de récrire dans la seconde occurrence du pattern `[_ * _]` avec l'hypothèse `H4` et de simplifier le but courant.
Le mécanisme de réflection entre les propositions décidables et les booléens décrit plus haut est intégré au nouveau langage de tactique. Par exemple, étant donné un contexte avec une proposition `H` de type `a && b`, la tactique `move/andP : H => H` applique le lemme `andP` à `H` et introduit dans le contexte une hypothèse `H` de type `a /\ b`. En revanche, lorsque le but est de la forme `a && b`, la tactique `apply/andP` change le but par `a /\ b`. Enfin, lorsque le but est de la forme `(a && b) -> G`, la tactique `case/andP => H1 H2` change le but en `G` et introduit deux hypothèses `H1 : a` et `H2 : b`.

Structures

Des bibliothèques de base sont définies dans SSReflect. C'est une hiérarchie de structure pour travailler avec les théories décidables et en particulier les types finis. La structure `eqType` définit les types munis d'une égalité décidable et équivalente à celle de Leibniz.

```
Structure eqType : Type := EqType {
  sort :> Type;
  _ == _ :  sort -> sort -> bool;
  eqP :  forall x y, reflect (x = y) (x == y)
}.
```

Le symbole `: >` déclare `sort` comme une coercion d'un `eqType` vers son type porteur. C'est une forme d'héritage. La structure `eqType` ne suppose pas seulement l'existence d'une égalité décidable `==`, en plus elle injecte cette égalité vers celle de Leibniz avec le proposition `eqP`. Nous pouvons ainsi profiter de la puissance de réécriture de Coq.
Une propriété majeure des structures d'`eqType` est qu'elles donnent la propriété de la *proof-irrelevance* pour les preuves d'égalités de leurs éléments. Ainsi il n'y a qu'une seule preuve de l'égalité pour chaque paire d'objets égaux.

```
Lemma eq_irrelevance: forall (d: eqType) (x y: d) (E: x = y) (E': x = y), E = E'.
```

Un ensemble sur une structure d'`eqType` est représenté par sa fonction caractéristique :

```
Definition set (d: eqType) := d -> bool.
```

Une propriété booléenne sur un `eqType` correspond donc à l'ensemble des éléments qui la satisfont. Avec cette définition les opérations ensemblistes comme l'intersection ou le complémentaire se définissent avec les fonctions booléennes correspondantes.

```
Definition setI (a b : set d) : set d := fun x => a x && b x.
Definition setC (a : set d) : set d := fun x => ~~ a x.
```

Il est utile d'avoir un type des listes sur un `eqType d` pour pouvoir définir plus naturellement des opérations qui se basent sur un test booléen comme la recherche d'un élément dans une liste. Le type des listes sur un `eqType d` se définit de façon inductive par :

```
Inductive seq : Type := Seq0 | Adds (x : d) (s : seq).
```

`Adds` et `Seq0` correspondent respectivement aux constructeurs `cons` et `nil` du type standard `list` de Coq. Le type `seq d` définit les listes sur un `eqType d`. Sur ce nouveau type de liste des fonctions sont définies pour manipuler et raisonner sur ses éléments. La fonction `foldr` correspond dans SSReflect à l'opération *fold* utilisée en programmation fonctionnelle. Elle est définie par récurrence sur une liste. L'opération d'extraction d'un élément d'une liste est donnée par la fonction `sub`. Par exemple `sub x0 s i` retourne l'élément d'indice `i` de la liste `s`, si `i` est strictement inférieur à la longueur de la liste, et `x0` dans le cas contraire. La fonction `mkseq` permet de construire une liste de longueur donnée à partir d'une fonction sur les entiers. Par exemple, `mkseq f n` correspond à la liste `[(f 0), (f 1), ..., (f n-1)]`.

Un type fini peut être vu comme un ensemble fini. Il peut alors être représenté par une liste de tous ses éléments. La définition du type liste sur un `eqType` est à la base de celle des types finis. La structure `finType` se compose d'une liste sur un `eqType` et de la preuve qu'aucun élément de cette liste n'apparaît plus d'une fois.

```
Structure finType : Type := FinType {
  sort :> eqType;
  enum :  seq  sort;
  enumP :  forall x, count (set1 x) enum = 1
}.
```

Dans cette définition `(set1 x)` est l'ensemble singleton x et `(count f l)` calcule le nombre d'éléments `y` de la liste `l` pour lesquels `(f y)` est vraie. Le paramètre `enum` correspond à la liste des éléments du type fini. Par exemple pour un `finType d`, `(enum d)` retourne la liste des éléments de `d`.

Pour représenter les types finis des éléments de l'intervalle $0..n-1$, la bibliothèque SSReflect fournit une famille de types nommée `ordinal` dont les éléments sont des paires composées d'un nombre entier p et d'une preuve que p est inférieur à n. Comme cette preuve est basée sur un test booléen, la propriété d'irrelevance s'applique et les éléments de ce type sont uniquement caractérisés par la composante p. La notation `I_(n)` désigne le type `ordinal n`.

4. Formalisations Coq

Dans ce travail de formalisation du théorème de Cayley-Hamilton, nous utilisons des bibliothèques sur les opérations indexées et les déterminants. Ces bibliothèques ont été développées par Y. Bertot et G. Gonthier dans le cadre du projet "Mathematical Components". La bibliothèque sur les polynômes fournit une formalisation des propriétés algébriques des polynômes, du morphisme d'évaluation et du théorème du reste. La définition de l'isomorphisme entre l'anneau des matrices de polynômes et celui des polynômes de matrices est l'étape ultime de la formalisation du théorème de Cayley-Hamilton.

4.1. Les opérations indexées

Dans la définition des opérations sur les matrices, par exemple la multiplication ou le calcul du déterminant, les opérations indexées (somme et produit) sont fréquentes. Factoriser la preuve de propriétés générales sur les sommes et produits indexés permet de réduire considérablement la longueur et la complexité des preuves et d'avoir des énoncés plus lisibles. Une bibliothèque pour les opérations indexées n'est pas seulement utile dans le développement sur la théorie des matrices, elle pourra l'être aussi dans des développements plus généraux que l'algèbre linéaire.

Une opération indexée est la généralisation de la définition d'une opération binaire aux éléments d'une suite finie. Dans le cas particulier de l'addition, c'est la somme de tous les éléments d'une suite donnée. L'opération indexée est définie par :

```
Definition reducebig R I op nil (r : seq I) P F : R :=
  foldr (fun i x => if P i then op (F i : R) x else x) nil r.
```

La fonction `reducebig` a comme paramètres un type quelconque `R`, un `eqType I`, une opération binaire `op` sur `R`, un élément `nil` de `R` correspondant à l'ensemble vide, une liste `r` sur `I`, une propriété caractéristique `P` sur `I` (une fonction de `I` vers `bool` : un ensemble sur `I`) et une fonction `F` de `I` vers `R`. Le résultat de `reducebig` correspond schématiquement à :

$$F\ p_1\ op\ F\ p_2\ op\ \cdots\ op\ F\ p_n\ op\ nil,$$

Les p_i sont les éléments de la liste r pour lesquels la propriété P est vraie : les éléments de l'ensemble P. L'utilisation de `reducebig` est plus naturelle lorsque l'opération est associative et commutative et lorsque `nil` est l'élément neutre de cette opération. En d'autres termes, lorsque `(R, op, nil)` est un monoïde.
La notation `\big[*%M/1]_(i| P i) F i` correspond à l'application de `reducebig` à une opération `*` d'un monoïde qui a pour élément neutre `1`. Le reste des paramètres est inféré de façon implicite par Coq. Par exemple `\big[*%M/1]_(i| i < n) i` équivaut à la notation mathématique $\sum_{i<n} i$.

Un lemme intéressant sur `reducebig` est celui qui permet d'effectuer l'opération usuelle de réindexation. Dans le cas d'une somme indexée, ce lemme correspond à l'égalité entre les sommes $\sum_{i=0}^{n}(i+m)$ et $\sum_{j=m}^{n+m} j$. Une façon de formaliser cette égalité est de considérer que i et j sont de types différents, respectivement $[0..n]$ et $[m..n+m]$, et qu'il existe une bijection entre ces deux types.

Cette bijection est la fonction $f : x \to x + m$.
Le prédicat `ibjective P h` permet de dire que la fonction `h` est bijective sur l'ensemble `P`. Le lemme de ré-indexation s'énonce alors comme suit :

```
Lemma reindex : forall (I J : finType) (h : J -> I) P F,
  ibijective P h ->
  \big[*%M/1]_(i | P i) F i = \big[*%M/1]_(j | P (h j)) F (h j).
```

Un autre résultat intéressant sur les opérations indexées est celui qui permet de décomposer cette opération suivant une partition de l'ensemble d'indice. Par exemple, dans le cas d'une somme indexée, le résultat s'écrit $\sum_{i=0}^{n+m} i = \sum_{i=0}^{n} i + \sum_{i=n+1}^{n+m} i$. La généralisation de cette propriété peut s'écrire formellement :

```
Lemma bigID : forall (I : finType) (a : set I) (P : I -> bool) F,
  \big[*%M/1]_(i | P i) F i
    = \big[*%M/1]_(i | P i && a i) F i * \big[*%M/1]_(i | P i && ~~ a i) F i.
```

Dans ce lemme, étant donné un ensemble `a`, une partition d'un ensemble `P` est donnée par les deux ensembles $\texttt{P} \cap \texttt{a}$ et $\texttt{P} \cap \bar{\texttt{a}}$, où $\bar{\texttt{a}}$ est l'ensemble complémentaire de $\bar{\texttt{a}}$. La somme des éléments indexés par `P` peut être donc décomposée suivant ces deux ensembles.

4.2. Structures canoniques

Dans l'assistant de preuve Coq, le mécanisme des `Canonical Structure` permet de définir une instance d'un type enregistrement (`Record` ou `Structure`) qui pourra être utilisée lors du processus d'inférence de type dans des équations invoquant des arguments implicites. Par exemple, pour définir une structure de `eqType` sur le type `nat` il faut une égalité décidable sur les entiers et prouver que cette égalité est équivalente à celle de Leibniz sur les entiers.

```
Fixpoint eqn (m n : nat) {struct m} : bool :=
  match m, n with
  | 0, 0 => true
  | S m', S n' => eqn m' n'
  | _, _ => false
  end.
Lemma eqnP : reflect_eq eqn.
Proof.
...
Qed.
Canonical Structure nat_eqType := EqType (@eqnP).
```

Le lemme suivant montre un exemple simple d'utilisation des `Canonical Structure`.

```
Lemma eqn_add0 : forall m n:nat, (m + n == 0) = (m == 0) && (n == 0).
```

Rappelons que == dénote l'égalité dans un `eqType`. Dans cet énoncé Coq s'attend à ce que `m` et `n` soient d'un type `eqType`. Comme ils sont de type `nat`, Coq cherche alors une définition d'un `eqType` dont le paramètre `sort` est `nat`. Grâce à la définition de `Canonical Structure nat_eqType`, Coq peut inférer automatiquement le type `nat_eqType` aux arguments `m` et `n`.

Le mécanisme des `Canonical Structure` est puissant et très utile dans le travail avec les structures algébriques comme les groupes ou les anneaux. SSReflect contient une bibliothèque `ssralg` qui, en

utilisant ce mécanisme, fournit une hiérarchie de structures algébriques regroupant monoïde, groupe, anneau et corps. En utilisant cette bibliothèque, les définitions des types polynômes et matrices sont paramétrées par l'anneau de leurs coefficients. Avec l'utilisation des `Canonical Structure` sur ces types, Coq pourra inférer automatiquement la structure d'anneau correspondante. Ceci nous permet d'unifier les notations pour les opérations algébriques sur ces types (addition, multiplication et opposé) et d'avoir ainsi des énoncés proches de ceux utilisés en mathématique standard et plus lisible du point de vue de l'utilisateur.

4.3. Matrices et déterminants

Une matrice sur un anneau R est une liste de coefficients doublement indexée. Elle peut être vue comme une fonction qui associe à une position (i, j) une valeur dans l'anneau R. Étant donnés m et n deux entiers et R un anneau, une matrice sur R (un objet de type $M_{m,n}(R)$) peut être représentée par la fonction suivante : $[0..n[\to [0..m[\to R$. Pour définir un `eqType` sur les matrices, qui sont des fonctions, nous avons besoin de l'extensionnalité. La fonction `fgraphType` construit le graphe des fonctions dont le domaine est un `finType` et le co-domaine un `eqType`. Avec la définition des graphes de fonctions, les fonctions sont ainsi munies d'une égalité de Leibniz. Pour deux fonctions `f` et `g`, les notations `f =1 g` et `f =2 g` correspondent respectivement à $\forall$`x, f x = g x` et $\forall$`x y, f x y = g x y`. Dans le cas où ces deux fonctions ont des domaines de type `finType` et des co-domaines de type `eqType`, les notations précédentes sont équivalentes à `f = g`.

Le type des matrices de taille (m, n) est défini par :

```
Definition matrix (m n :nat) := fgraphType (I_(m) * I_(n)) R.
```

Les fonctions `matrix_of_fun` et `fun_of_matrix` permettent respectivement de définir un objet de type `matrix` à partir d'une fonction et de convertir un objet de type `matrix` en une fonction à deux arguments. Cette dernière n'est autre qu'une coercion du type `matrix` vers celui des fonctions. Elle nous permet de dire que deux matrices `A` et `B` sont égales si et seulement si nous avons `A =2 B`. Ce qui veut dire que leurs fonctions associées sont égales : `fun_of_matrix A =2 fun_of_matrix B`.
Dans la suite, les notations `M_(n)` et `\Z x` correspondent respectivement au type des matrices carrées et à la matrice scalaire en `x`. La notation `\matrix_(i,j) E`, où `E` est une expression en `i` et `j`, correspond à l'application de `matrix_of_fun` à la fonction `f i j => E i j`. Par exemple, la matrice scalaire correspondant à un élément `x` est donnée par la formule suivante :

```
Definition scalar_mx n x : M_(n) := \matrix_(i, j) (if i == j then x else 0).
```

La bibliothèque sur les déterminants utilise la formule de Leibniz pour définir le déterminant. Ce choix est motivé par le fait que cette formule est bien adaptée et que nous disposons des "ingrédients" nécessaires à sa formalisation. En effet, une formalisation des permutations (groupe, signature ...) a été déjà développée dans le cadre du travail sur les groupes finis [8]. Etant donnée une matrice carrée A de dimension n, le déterminant est défini par :

$$\det(A) = \sum_{\sigma \in S_n} \epsilon(\sigma) \prod_{i=1}^{n} a_{i,\sigma(i)} \tag{5}$$

Dans cette formule, il est question d'opérations indexées pour les sommes et les produits, de groupe de permutations (S_n) et de signature de permutations$(\epsilon(\sigma))$. Les notations mathématiques cachent plusieurs autres éléments. Dans la bibliothèque sur les déterminants, pour pouvoir formaliser la formule (5) :

- l'indexation des lignes et colonnes de la matrice par des entiers est remplacée par une indexation par les éléments du type `I_(n)`,
- l'ensemble des permutations sur cet ensemble fini est décrit comme un ensemble fini qui pourra être énuméré et donc servir d'ensemble d'indices.

Avec ces choix, grâce aux développements sur le calcul de la parité des permutations et à celui sur les groupes de permutations, la formule (5) s'écrit :

```
Definition determinant n (A : M_(n)) :=
  \sum_(s : S_(n)) (-1) ^ s * \prod_(i) A i (s i).
```

Les notations `\sum` et `\prod` représentent respectivement la somme et le produit indexés. Ce sont des instances de `reducebig` pour les opérations internes (addition et multiplication) de l'anneau des coefficients de la matrice. La notation `S_(n)` représente le groupe des permutations sur un ensemble à n éléments. Dans la suite, la notation `\det` représentera la fonction `determinant`.

Pour exprimer la règle de Cramer, la co-matrice d'une matrice est définie à l'aide de la fonction `row'`. Cette dernière prend en entrée un nombre i inférieur à m (un élément de type `I_(m)`) et une matrice de taille (m, n) ; elle retourne la matrice $(m-1, n)$ où la rangée i a été enlevée. Avec les mêmes arguments, la fonction `row` retourne la matrice $(1, n)$ (une rangée et n colonnes) qui contient la rangée i. La transposée de la co-matrice est représentée par la fonction `adjugate`.

```
Definition cofactor n (A : M_(n)) (i j : I_(n)) :=
   (-1) ^+(i + j) * \det (row' i (col' j A)).
Definition adjugate n (A : M_(n)) := \matrix_(i, j) (cofactor A j i).
```

L'égalité de Cramer est alors formellement représentée par le lemme `mulmx_adjr` :

```
Lemma mulmx_adjr : forall n (A : M_(n)), A * adjugate A = \Z (\det A).
```

La preuve utilise la formule de Laplace ($\det(A) = \sum_{i=1}^{n} a_{i,j}\text{cofacteur(A)}_{i,j}$) qui donne le déterminant en fonction des coefficients d'une seule ligne ou colonne et des cofacteurs correspondants. Celle-ci est formellement donnée par :

```
Lemma expand_determinant_row : forall n (A : M_(n)) i,
  \det A = \sum_(j) A i j * cofactor A i j.
```

Le lemme selon lequel le déterminant est une forme alternée (le déterminant d'une matrice, où au moins deux lignes sont identiques, est nul) s'énonce formellement comme suit :

```
Lemma alternate_determinant : forall n (A : M_(n)) i1 i2,
  i1 != i2 -> A i1 =1 A i2 -> \det A = 0.
```

Comme les matrices sont des fonctions à deux arguments, le terme `(A i1)` est une fonction à un argument et il correspond à la ligne d'indice `i1` de la matrice `A`.

4.4. Polynômes

Un polynôme est défini par la liste de ses coefficients a_i qui appartiennent à un anneau R :

$$a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$$

Cette représentation n'est malheureusement pas unique, en effet les polynômes $a_n x^n + a_{n-1}x^{n-1} + \cdots + a_1 x + a_0$ et $0x^{n+1} + a_n x^n + a_{n-1}x^{n-1} + \cdots + a_1 x + a_0$ ont des listes de coefficients différentes mais représentent le même polynôme. Pour avoir une égalité de Leibniz pour cette représentation, il est nécessaire de ne considérer que les polynômes normalisés, c'est-à-dire ceux dont le coefficient de plus grand degré est non nul, et avoir une égalité de Leibniz sur les coefficients. Les polynômes sont donc représentés par la structure suivante :

```
Structure polynomial (R : ring) : Type := Poly {
  p :>  seq R;
  normal : last 1 p != 0
}.
```

La propriété `normal` dit que le dernier élément de la liste des coefficients est non nul. Avec cette définition, nous pouvons donc définir une structure de `eqType` sur les polynômes. Nous avons défini la fonction coefficient des polynômes par :

```
Definition coef (p : polynomial) i := sub 0 p i.
```

La fonction `coef p` est de type `nat -> R`. Le lemme suivant permet d'avoir l'équivalence entre l'égalité entre les polynômes et celles entre les fonctions de coefficient :

```
Lemma coef_eqP : forall p1 p2, coef p1 =1 coef p2 <-> p1 = p2.
```

Avec ce lemme, nous pouvons passer de notre représentation structurelle des polynômes vers celle qui ne considère que la fonction des coefficients. L'avantage de la second représentation est de rendre les preuves des propriétés algébriques des polynômes plus intuitives. Par exemple, la multiplication de deux polynômes est définie par :

$$\left(\sum_{i=0}^{n} a_i x^i\right)\left(\sum_{j=0}^{m} b_j x^j\right) = \sum_{k=0}^{m+n}\left(\sum_{i+j=k} a_i b_j\right) x^k. \tag{6}$$

La preuve de l'associativité de cette multiplication se ramène à des raisonnements sur des sommes indexées, sans avoir besoin de faire des récurrences sur les polynômes.
Dans la suite, les notations `\X` et `\C c` correspondent respectivement au monôme x et au polynôme constant c.
L'opération de multiplication d'un polynôme par `x` (décalage à droite) et addition d'une constante est l'une des opérations de base sur les polynômes. Dans la bibliothèque elle est réalisée par la fonction suivante :

```
Definition horner c p : polynomial :=
  if p is Poly (Adds _ _ as s) ns then Poly (ns : normal (Adds c s)) else \C c.
```

A partir de cette définition, la fonction de construction d'un polynôme à partir d'une liste de coefficients se définit simplement par :

```
Definition mkPoly := foldr horner \C0.
Notation "\poly_ ( i < n ) E" := (mkPoly (mkseq (fun i : nat => E) n)).
```

La notation `\poly` permet de construire un polynôme à partir d'une fonction de coefficients. Par exemple, le polynôme correspondant à `\poly_ ( i < n ) i` est : $n - 1x^{n-1} + \cdots + 1x + 0$

Les opérations de base sur les polynômes sont définies par récurrence sur la liste des coefficients. La liste résultat est ensuite normalisée par la fonction `mkPoly`. Par exemple, la multiplication de deux polynômes est définie comme suit :

```
Fixpoint mult_poly_seq (s1 s2 : seq R) {struct s1} : seq R :=
  if s1 is Adds c1 s1' then
    add_poly_seq (maps (fun c2 => c1 * c2) s2)
                 (Adds 0 (mult_poly_seq s1' s2))
  else seq0.

Definition mult_poly (p1 p2 : polynomial) := mkPoly (mult_poly_seq p1 p2).
```

Dans la seconde définition, la conversion de type entre les types `polynomial` et `seq` permet d'écrire `mult_poly_seq p1 p2` bien que `p1` et `p2` sont de type `polynomial`. Le lemme `coef_mul_poly`

```
Lemma coef_mul_poly : forall p1 p2 i,
  coef (mult_poly p1 p2) i = \sum_(j <= i) coef p1 j * coef p2 (i - j).
```

donne une relation entre les coefficients de deux polynômes et ceux du résultat de leur multiplication. Il correspond à la relation de la formule (6).

Une autre opération importante sur les polynômes est la fonction d'évaluation d'un polynôme. Elle consiste à remplacer sa variable par une valeur donnée. Cette fonction peut être décrite avec le schéma de Horner pour un polynôme p et une valeur x par :

$$p(x) = ((...((a_n x + a_{n-1})x + a_{n-2})x + ...) + a_1)x + a_0 \tag{7}$$

Suivant le schéma (7) l'évaluation ne dépend que de la liste des coefficients et de la valeur où nous évaluons. Elle se définit par récurrence sur la liste des coefficients. La fonction d'évaluation peut être définie par récurrence sur une liste arbitraire comme suit :

```
Fixpoint eval_poly_seq (s : seq R) (x : R) {struct s} : R :=
  if s is (Adds a s') then eval_poly_seq s' x * x + a else 0.
```

Rappelons que dans la définition de la structure `polynomial` nous avons une coercion entre elle et le type de la liste des coefficients. Ceci nous permet de définir l'évaluation d'un polynôme de la manière suivante :

```
Definition eval_poly (p : polynomial R) : R-> R := eval_poly_seq p.
```

La notation `p.[c]` correspond à l'application de la fonction `eval_poly` en `p` et `c`.
Les propriétés de morphisme de la fonction d'évaluation sont utilisées implicitement dans la preuve du théorème de Cayley-Hamilton. Ces propriétés sont données par les lemmes suivants :

```
Lemma eval_polyC : forall c x, (\C c).[x] = c.
Lemma eval_poly_plus : forall p q x, (p + q).[x] = p.[x] + q.[x].
Lemma eval_poly_mult : forall p q x, x * q.[x] = q.[x] * x ->
                                     (p * q).[x] = p.[x] * q.[x].
```

Dans le lemme `eval_poly_mult` sur l'évaluation d'un produit de polynômes, il est nécessaire d'avoir que la valeur `x` où l'on évalue le polynôme `p * q` commute avec l'évaluation de `q` en cette même valeur. Cette hypothèse est nécessaire vue que l'on évalue des polynômes sur un anneau non commutatif : l'anneau des matrices.

Après ces développements, le théorème du reste peut s'énoncer comme suit :

```
Theorem factor_theorem : forall p c,
  reflect (exists q, p = q * (\X - \C c)) (p.[c] == 0).
```

Dans la preuve de ce théorème, pour pouvoir dire que `p.[c]` est égale à `q.[c] * (\X - \C c).[c]`, il faut prouver que les coefficients du polynôme `(\X - \C c)` commutent avec `c`. Ce qui se prouve facilement car `1` et `c` commutent avec `c`.

4.5. Preuve de Cayley-Hamilton

Le morphisme entre l'anneau des matrice de polynômes et celui des polynômes de matrices est la partie centrale de la preuve du théorème de Cayley-Hamilton. Les autres composantes de la preuve, la règle de Cramer et le théorème de factorisation, sont des propriétés qui se rattachent respectivement aux matrices et aux polynômes.
Ce morphisme que nous allons appeler `phi` prend en argument une matrice de polynômes et retourne un polynôme de matrices. La longueur de la liste des coefficients du polynôme résultat est la taille maximale des polynômes de la matrice de départ. La taille d'un polynôme correspond à la longueur de la liste de ces coefficients, en d'autre terme son degré plus un. Pour une matrice de polynômes `A`, `phi A` est le polynômes de matrices dont le coefficient d'indice `k` est la matrice dont le coefficient en `i` et `j` est `coef (A i j) k`. Dans la suite, les notations `R[X]`, `M(R)`, `M(R[X])` et `M(R)[X]` représentent respectivement l'anneau des polynômes, celui des matrices, celui des matrices de polynômes et celui des polynômes de matrices.

```
Definition phi (A : M(R[X])) : M(R)[X] :=
  \poly_(k < \max_(i) \max_(j) size (A i j)) \matrix_(i, j) coef (A i j) k.
```

Le lemme `coef_phi` permet d'exprimer la relation entre une matrice de polynômes et son image par `phi`.

```
Lemma coef_phi : forall A i j k, coef (phi A) k i j = coef (A i j) k.
```

Pour pouvoir définir l'évaluation d'une matrice en son polynôme caractéristique, nous avons défini l'injection de l'anneau des polynômes vers celui des polynômes de matrices.

```
Definition Zpoly (p : R[X]) : M(R)[X] := \poly_(i < size p) \Z (coef p i).
```

Le polynôme caractéristique d'une matrice est définit en appliquant la définition du déterminant à la matrice de polynômes $xI_n - A$.

```
Definition matrixC (A : M(R)) : M(R[X]) := \matrix_(i, j) \C (A i j).
Definition char_poly (A : M(R)) : R[X] := \det (\Z \X - matrixC A).
```

La fonction `matrixC` est l'injection canonique de l'anneau des matrices vers celui des matrices de polynômes.

Après ces définitions, le théorème de Cayley-Hamilton est prouvé formellement de la façon suivante :

```
Theorem Cayley_Hamilton : forall A, (Zpoly (char_poly A)).[A] = 0.
Proof.
move=> A; apply/eqP; apply/factor_theorem.
rewrite -phi_Zpoly -mulmx_adjl phi_mul; move: (phi _) => q; exists q.
by rewrite phi_add phi_opp phi_Zpoly phi_polyC ZpolyX.
Qed.
```

La preuve se déroule exactement comme décrit dans la seconde section. Après avoir appliqué le théorème du reste, le polynôme facteur est donné en récrivant avec la règle de Cramer (`mulmx_adjl`). Le résultat du théorème de Cayley-Hamilton est alors prouvé par des réécritures et simplifications dans le terme obtenu. L'utilisation de SSReflect, des mécanismes des `Canonical Structure` et des notations, ainsi que la définition hiérarchique des structures de données ont permis d'aboutir à une preuve aussi concise : 3 lignes de codes.

5. Conclusion

Nous avons présenté une formalisation du théorème de Cayley-Hamilton qui adopte une approche modulaire. La preuve que nous avons présentée dans la section 4.4 peut paraître très simple ; mais la conception a été assez longue que ce soit pour choisir l'architecture globale de la preuve ou le bon type de données pour représenter les structures manipulées (polynômes et matrices). Les choix ont été motivés par des soucis de lisibilité et de réutilisabilité. L'utilisation des Canonical Structure de Coq nous a permis d'avoir des énoncés proches de ceux utilisés en mathématiques usuelles. Le découpage des différentes composantes de la preuve sous forme modulaire (les opérations indexées, les matrices et les polynômes) favorise la réutilisation de ces bibliothèques dans des développements indépendants. Les bibliothèques sur les opérations indexées et les matrices seront réutilisées dans nos prochains travaux sur la théories des caractères, une des composantes de la preuves du théorème de Feit-Thompson.
Ce travail que nous avons présenté ici est la première formalisation du théorème de Cayley-Hamilton. Ce n'est pas la première formalisation des matrices ou des polynômes. Des formalisations de ces structures sont présentées respectivement dans [9, 11] et [5, 10]. Mais c'est le premier développement qui regroupe une formalisation des matrices et polynômes et où ces objets sont assemblées pour former de nouveaux objets : les matrices de polynômes et les polynômes de matrices.

Dans la formalisation du théorème de Cayley-Hamilton, présentée dans cet article, nous avons choisi de construire nos structures de données sur des types munis d'une égalité décidables : les `eqType`. En plus du fait qu'en mathématiques classiques tous les types sont décidables, notre preuve sur les types où l'égalité est décidables peut être généralisés vers les types quelconques. Ceci se fait en remarquant que tout anneau est une $\mathbf{Z}$-algèbre et en considérant le morphisme d'évaluation des polynômes à n^2 variables et à coefficients dans $\mathbf{Z}$ qui est un type où l'égalité est décidable. Il va falloir alors travailler avec les `Setoid`.

Dans ce développement la bibliothèque sur les polynômes comprend 83 objets (définitions et lemmes) pour environs 490 lignes de codes. Les définitions et lemmes propres à la preuve du théorème de Cayley-Hamilton sont au nombre de 15 pour 125 lignes de codes. Les sources du développement sont disponibles à l'adresse suivante : `http://www-sop.inria.fr/marelle/Sidi.Biha/cayley/`.

Références

[1] Paul et Jack Abad, *The Hundred Greatest Theorems*, Disponible à `http://personal.stevens.edu/~nkahl/Top100Theorems.html`.

[2] Jeremy Avigad, Kevin Donnelly, David Gray, et Paul Raff, *A Formally Verified Proof of the Prime Number Theorem*, ACM Transactions on Computational Logic, A paraître.

[3] Yves Bertot, Pierre Castéran, *Interactive Theorem Proving and Program Development Coq'Art : The Calculus of Inductive Constructions*, Springer Verlag, 2004.

[4] Nathan Jacobson, *Lectures in Abstract Algebra : II. Linear Algebra*, Springer Verlag, 1975.

[5] Herman Geuvers, Freek Wiedijk et Jan Zwanenburg, *A Constructive Proof of the Fundamental Theorem of Algebra without Using the Rationals*, Types for Proofs and Programs, TYPES 2000 International Workshop, Selected Papers, volume 2277 of LNCS, pages 96-111, 2002.

[6] Georges Gonthier, *A computer-checked proof of the four-colour theorem*, Disponible à `http://research.microsoft.com/~gonthier/4colproof.pdf`.

[7] Georges Gonthier, Assia Mahboubi, *A small scale reflection extension for the Coq system*, Disponible à `http://www.msr-inria.inria.fr/~assia/rech-eng.html` (à paraître comme RR Inria).

[8] Georges Gonthier, Assia Mahboubi, Laurence Rideau, Enrico Tassi et Laurent Théry, *A Modular Formalisation of Finite Group Theory*, Rapport de Recherche 6156, INRIA, 2007.

[9] Nicolas Magaud, *Ring properties for square matrices* contribution à Coq, `http://coq.inria.fr/contribs-eng.html`.

[10] Piotr Rudnicki, *Little Bezout Theorem (Factor Theorem)*, Journal of Formalized Mathematics volume 15, 2003, Disponible à `http://mizar.org/JFM/Vol15/uproots.html`.

[11] Jasper Stein, *Linear Algebra* contribution à Coq, `http://coq.inria.fr/contribs-eng.html`.

[12] Coq Team, *The Coq reference manual V 8.1*, `http://coq.inria.fr/V8.1/refman/index.html`.

[13] Freek Wiedijk, *Formalizing 100 Theorems*, `http://www.cs.ru.nl/freek/100/`.

A Formal Verification for Kantorovitch's Theorem

Ioana Paşca [1]

1: INRIA Sophia Antipolis
Ioana.Pasca@sophia.inria.fr

Abstract

Kantorovitch's theorem gives sufficient conditions for the convergence of Newton's method. We present a full formalization of this theorem in the case of a real function. The work is accomplished inside the Coq proof assistant and it is based on the `Reals` library provided by the theorem prover. For the general case of the theorem we first describe a way to represent concepts from multivariate analysis, as Coq does not offer such a library. We then discuss the proof of Kantorovitch's theorem based on this representation.

1. Introduction

The main purpose of formal methods is to guarantee that software satisfies its formal specification. We are interested in extending such techniques to areas of computer science that rely on numerical methods. This is of interest because of the safety critical domains that rely on these methods like on-board software for planes, trains, real time programs used in medicine etc. Formal verifications of numerical methods are rare. We can mention the work of Micaela Mayero [19] in which real analysis concepts are formalized in order to prove the correctness of a differentiation algorithm.

At a slightly different level, we are interested in having mathematical concepts formalized inside proof assistants. In the long run, the aim would be to use them for extensive mathematical developments. At present, computer algebra systems like Maple or Matlab are more widely used than proof assistants, for representing mathematical concepts on machines. However there are cases where the level of accuracy provided by these systems is not sufficient and we would like to have a certification that the computation is indeed correct [16, 5].

In this context, we are interested in Newton's process, which approximates the root of a given function. Kantorovitch's theorem gives sufficient conditions for the convergence of this process.

A formal verification of such a theorem is of interest for computer scientists who use numerical methods in their developments. Newton's method is widely used because of its quadratic convergence rate.

Applying formal methods to a domain means encoding the concepts of that domain inside a proof assistant and verifying the desired properties within this setting. It is thus obvious that the success of formal verification for numerical methods depends on having an extensive and practical formalization of real analysis concepts inside the proof assistant. For the formal proof of Kantorovitch's theorem, we organized our work by starting with the case of a single variate real function and then generalizing for several dimensions. In section 2 we provide a survey of formal real analysis concepts in various theorem provers. We also present an outline of the mathematical proof of Kantorovitch's theorem. In section 3, we present the proof of the theorem in one dimension. Section 4 describes the multivariate analysis concepts we formalized and section 5 discusses the proof of the theorem in $\mathbb{R}^p$. The conclusions and possible extensions of this work are detailed in section 6.

2. Survey of related work

Mathematical proofs cannot be entirely discovered by mechanical means, but the correctness of a formal proof can be verified by a machine. Work in this domain has uncovered several approaches to representing the proofs and each of these approaches has led to a different system. Some of the most important proof systems at present are HOL [13], PVS [25], ACL2 [17], Isabelle[23] and Coq [2, 1]. Each of them offers a library of formalized mathematical concepts and verified theorems.

2.1. Formalizing real analysis concepts

As stated before we are interested in the way real analysis concepts are encoded to have a representation that is both feasible and corresponds to the mathematical concept being formalized. We are also interested in seeing what sort of theorems and results have been obtained using these concepts.

The first thing is to find a way to represent the real numbers. They can either be given an axiomatic definition as a complete ordered field satisfying the least upper bound principle or they can be defined constructively and proved the right properties. There are several constructions for the reals, the most famous being the Dedekind model, based on the notion of cut, the Cantor model, which uses Cauchy sequences of rational numbers and the Weierstrass model which uses decimal fractions.

The next step is to see how real analysis concepts can be efficiently formalized inside the proof assistant. There are two main approaches. One is to follow classical analysis where concepts are expressed using the usual $\varepsilon - \delta$ definitions. The other is to use non-standard analysis as first introduced by Robinson [24].

Non-standard analysis works with ideally small and ideally large numbers, so it formalizes the concepts of infinite and infinitesimal. It introduces numbers systems such as hyper-reals and the infinitely close relation, which help express concepts such as limit, continuity etc. The main argument in favor of using non-standard analysis is that, by using this "infinitely close relation", the manipulation of objects (e.g. derivatives, limits of functions) becomes algebraic and therefore theorems are easier to automate.

For HOL, the main work can be found in [14]. The real numbers are constructed using an adaptation of Cantor's method. Real analysis is dealt with in a classical way. One interesting fact is the way limits are implemented. As opposed to other systems, which treat independently the limit of a sequence and that of a function, HOL describes them using one concept by relying on the theory of nets. Results are achieved around continuity, differentiation, integrability and transcendental functions. We mention that a quantifier elimination procedure has also been implemented for this theory. Some applications of the developed theory involve verifications for floating point algorithms.

In Isabelle one can actually find most of the concepts formalized in both classical and non-standard analysis. What is interesting is the proof of equivalence between the concepts in the two approaches[8].

The ACL2 proof assistant has a non-standard approach to real analysis. [9] offers an introduction to non-standard analysis techniques and shows how they can be used to reason mechanically about concepts like transcendental functions. The formalization of continuity, differentiability etc. allows proving theorems such as intermediate value theorem and Rolle's theorem.

[6] presents an implementation of basic real analysis building on the axiomatic definition of the reals in the PVS theorem prover. This implementation includes definitions of convergence, continuity, differentiability of real-valued functions and proofs for theorems around this concepts (e.g. the mean value theorem).

In Coq, two approaches have been explored. The Coq Standard Library called `Reals` provides an axiomatic definition of the reals. The library contains a lot of results for real analysis: sequences and series, transcendental function, concepts of limit, continuity, differentiation, integration, calculus

theorems like the mean value theorem,the fundamental theorem of calculus etc.

There also exists a constructive formalization of real analysis in Coq. In C-CoRN (Coq Constructive Repository at Nijmegen [4]) the reals are build as a Cauchy completion of the rationals [22]. Among the most important work in this setting we can mention the constructive formalization of the fundamental theorem of algebra [10] and the fundamental theorem of calculus [3].

Closely related to our work, we can mention the existence in C-CoRN of a formalization for a root finding algorithm. The result is part of a larger project that dealt with the formalization of the Fundamental Theorem of Algebra [10] using a proof given by Kneser.

Our work was done using the Coq Proof Assistant and the Standard Library, i.e. the axiomatic, classical definition for real analysis concepts. For the second part of our work we also used the `ssreflect` extension and its libraries for reasons to be discussed in Section 4.

2.2. Mathematical proof of Kantorovitch's theorem

As stated above, we are interested in having sufficient conditions for the convergence of Newton's process in the case of an equation system. The problem was studied by Willers, Sténine, Ostrowski, Kantorovitch and others. In what follows, we present a special case of Kantorovitch's theorem that expresses the convergence of Newton's method for a finite system of non-linear equations. It establishes the existence and uniqueness of a solution for the initial equation system.

The statement of the theorem according to [7] is as follows:

Theorem 1 *Consider a system of non-linear algebraic or transcendent equations $f(x) = 0$, where the vector function $f : \mathbb{R}^p \to \mathbb{R}^p$ has continuous first and second partial derivatives in a certain domain ω, i.e. $f(x) \in C^{(2)}(\omega)$. Let $x^{(0)}$ be a point contained in ω with its closed ε-neighborhood $\overline{U_\varepsilon}(x^{(0)}) = \{\|x - x^{(0)}\| \leq \varepsilon\}$ also included in ω. If the following conditions hold:*

1. *the Jacobian matrix $W(x) = [\frac{\partial f_i(x)}{\partial x_j}]$ has an inverse for $x = x^{(0)}$, $\Gamma_0 = W^{-1}(x^{(0)})$ with $\|\Gamma_0\| \leq A_0$;*
2. *$\|\Gamma_0 f(x^{(0)})\| \leq B_0 \leq \frac{\varepsilon}{2}$;*
3. *$\sum_{k=1}^{p} |\frac{\partial^2 f_i(x)}{\partial x_j \partial x_k}| \leq C$ pour $i, j = 1, 2, ..., p$ and $x \in \overline{U_\varepsilon}(x^{(0)})$;*
4. *the constants A_0, B_0, C satisfy the inequality $2pA_0B_0C \leq 1$.*

then, for the initial approximation $x^{(0)}$, the Newton process

$$x^{(n+1)} = x^{(n)} - W^{-1}(x^{(n)})f(x^{(n)}) \tag{1}$$

$(n = 1, 2, ...)$ converges and the limit vector $x^* = \lim_{n\to\infty} x^{(n)}$ is a solution of the initial system, so that $\|x^* - x^{(0)}\| \leq 2B_0 \leq \varepsilon$. Moreover, in the domain $\{\|x - x^{(0)}\| \leq 2B_0\}$ the solution is unique.

Here is the outline of the proof for the theorem as presented in [7]:

- prove a collection of properties for each element of the Newton sequence;
- infer that it is a Cauchy sequence;
- use the completeness of $\mathbb{R}^p$ to prove the convergence;
- prove that the limit of the sequence is a root of the given function;
- prove that in a certain interval the root is unique.

For the first part, the properties verified for each element of the sequence are similar to those for $x^{(0)}$. Resoning by induction we get the following:

$$W(x^{(n)}) \text{ is invertible}$$
$$\|W^{-1}(x^{(n)})\| \leq A_n \tag{2}$$
$$\|W^{-1}(x^{(n)})f(x^{(n)})\| \leq B_n \leq \frac{\varepsilon}{2^{n+1}} \tag{3}$$
$$2pA_nB_nC \leq 1$$

where

$$A_n = 2A_{n-1}$$
$$B_n = pA_{n-1}B_{n-1}^2C$$

We are able to establish:

$$\overline{U}_{\varepsilon}(x^{(0)}) \supset \overline{U}_{\frac{\varepsilon}{2}}(x^{(1)}) \supset \ldots \supset \overline{U}_{\frac{\varepsilon}{2^n}}(x^{(n)}) \supset \ldots \tag{4}$$

From (4) we can infer that $x^{(n)}$ is a Cauchy sequence:

$$x^{(n+m)} \in \overline{U}_{\frac{\varepsilon}{2^n}}(x^{(n)}) \Rightarrow \|x^{(n+m)} - x^{(n)}\| \leq \frac{\varepsilon}{2^n}$$

The latter quantity can be made arbitrary small for $n > N$ and $m \in \mathbb{N}$, which is equivalent to Cauchy's criterion. We use the result that $\mathbb{R}^p$ is a complete metric space to deduce that the sequence converges. By taking the limit in (1) we get that the limit of the sequence is a root of function f.

To prove the uniqueness of the solution, we suppose that there exists another solution of the equation and prove that it is also the limit of the sequence. By uniqueness of this limit we have the desired result.

The reasoning steps behind (2) or (3) use non trivial results from matrix theory and differential calculus in several dimensions (e.g. Taylor's formula) as we shall point out again later on.

3. Proof in one dimension

3.1. Informal presentation

Kantorovitch's theorem gives the conditions of convergence in the general, multidimensional case. Naturally, it can be reduced to the unidimensional case. This entails some simplifications in the conditions imposed as well as in the proof. This adapted proof of Kantorovitch's theorem will later serve as a guideline in the general proof.

We also made an adjustment in the statement of the theorem by loosening the constraint that the function should be twice derivable. We just impose a bounding condition on the variation of the first derivative, which is trivially verified if the function does have a continuous second derivative (condition 3 in Theorem 2).

The modified statement of the theorem is as follows:

Theorem 2 *Consider an equation $f(x) = 0$, where $f : [a, b] \to \mathbb{R}$, a, $b \in \mathbb{R}$ $f(x) \in C^{(1)}([a, b])$. Let $x^{(0)}$ be a point contained in $[a, b]$ with its closed ε-neighbourhood $\overline{U_{\varepsilon}}(x^{(0)}) = \{|x - x^{(0)}| \leq \varepsilon\} \subset [a, b]$. If the following conditions hold:*

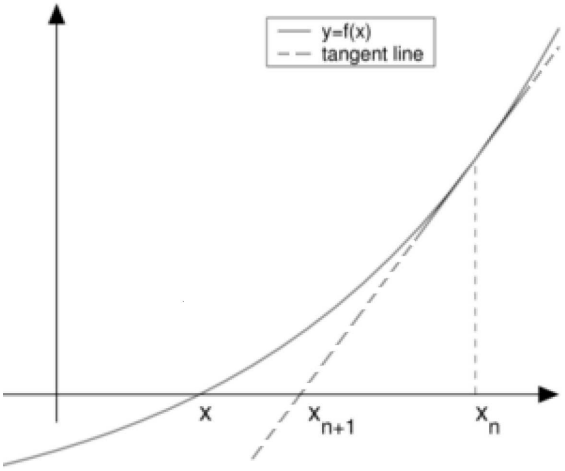


Figure 1: Newton's method

1. $f'(x^{(0)}) \neq 0$ *and* $|\frac{1}{f'(x^{(0)})}| \leq A_0$;
2. $|\frac{f(x^{(0)})}{f'(x^{(0)})}| \leq B_0 \leq \frac{\varepsilon}{2}$;
3. $\forall x, y \in [a, b], |f'(x) - f'(y)| \leq C|x - y|$
4. *the constants* A_0, B_0, C *satisfy the inequality* $2A_0B_0C \leq 1$.

then, for an initial approximation $x^{(0)}$*, the Newton process*

$$x^{(n+1)} = x^{(n)} - \frac{f(x^{(n)})}{f'(x^{(n)})} \tag{5}$$

($n = 1, 2, ...$*) converges and the limit vector* $x^* = \lim_{n\to\infty} x^{(n)}$ *is a solution of the initial system, so that* $|x^* - x^{(0)}| \leq 2B_0 \leq \varepsilon$*. Moreover, in the domain* $\{|x - x^{(0)}| \leq 2B_0\}$ *the solution is unique.*

The intuition behind these conditions can be more easily understood by analysing Figure 1. The conditions say that if the slope of the function is sufficiently steep and if the variation of this slope (i.e. the first derivative) is bounded sufficiently tight, then the successive approximations will get closer to the root each time, moreover, the process will converge to the root.

3.2. The formal proof

As stated before, Coq contains a library with results about real numbers. The formalization of reals in this standard library is axiomatic and done in classical logic. The library contains most of the results needed in the proof. The work accomplished was mainly conducted in two directions: providing results not found in the standard library `Reals` and proving the structure of the theorem. During this work, we also tried to understand where difficulties arise in such a development and what could be done to facilitate such proofs. An interesting issue was working with derivatives.

3.2.1. Derivation

Let $f : \mathbb{R} \to \mathbb{R}$ be a derivable function on $[a, b]$. "On paper" we can write the corresponding Newton's sequence $x^{(n+1)} = x^{(n)} - \frac{f(x^{(n)})}{f'(x^{(n)})}$, without worrying whether the term $x^{(n)}$ is in the interval $[a, b]$ (so

that we know the derivative actually exists). However, the formalization of derivatives in Coq requires that when one talks about the derivative of a function in a point, one has to provide a term that states that the function is actually derivable at that point. The goal is to ensure that derivatives are properly used. Nevertheless, this is an inconvenience when manipulating derivatives.

The way we worked around this particular impediment is by defining a total function to use in the definition of the sequence and then say that on interval $[a, b]$ this function is equal to the derivative of f. This enables us to define our concepts and at the same time prevents us from using properties of the derivative in a point before proving the function is derivable there.

3.2.2. Higher order derivatives

Another issue that for the moment remains unresolved is that of higher order derivatives. This is one of the reasons for which we chose to generalize the statement of the theorem. We did not want to talk about the second derivative because at this moment in Coq it can be referred to only as the derivative of the first derivative. This renders its use rather tedious. We would prefer having a general mechanism for dealing with derivatives.

3.2.3. Statement of the theorem

For the actual proof of the theorem we were able to follow rather accurately the "proof on paper". The Coq proof assistant possesses some tactics that help automatize steps in the proof dealing with equality relations (e.g `ring`, `field`), with inequalities on the real numbers (`fourier`) or with differentiation rules (`reg`). They made a big difference in the amount of work that had to be accomplished.

The statements of the main lemmas are given bellow. In the code, the `Variable` declarations describe objects that are assumed to exist, the `Hypothesis` declarations describe properties that are assumed to hold and the `Fixpoint` declaration defines the Newton sequence as a recursive function.

```
Variables a b:R. (*the ends of the interval*)
Variables f f':R→R. (*the function and its derivative*)
Variables A0 B0 C:R. (*the constants from the theorem*)
Variable X0:R. (*the initial approximation*)
Fixpoint Xn (n:nat): R := (*the Newton sequence*)
match n with
| 0 => X0
|S n => (Xn n) - (f (Xn n))/(f' (Xn n))
end.

(*the hypothesis on the function and the constants*)
Hypothesis pr: ∀ t:R, c_I a b t → derivable_pt f t.
Hypothesis Hder_f: ∀ (t:R)(H:c_I a b t), derive_pt f t (pr t H)=f' t.
Hypothesis Hcont_f':∀ y, (c_I a b y) → continuity_pt f' y.
Hypothesis Hlim_f':∀ y1 y2:R, c_I a b y1 → c_I a b y2 →
      Rabs (f' y1 - f' y2) ≤ c * (Rabs (y1-y2)).
Hypothesis Hincl:included (c_disc X0 eps) (c_I a b).
Hypothesis Hdif_f':f' X0 ≠ 0 .
Hypothesis Habs_a: Rabs (/(f' X0)) ≤ A0.
Hypothesis Habs_b:Rabs (f X0/(f' X0)) ≤ B0 .
Hypothesis A0_b0_c: 2*A0*B0*C≤1.

(*the theorem stating the existence*)
Theorem kantoro_exist_b:
∃ xs:R, Un_cv Xn xs ∧ c_disc X0 (2*b0) xs ∧ f xs = 0.
```

```
(*uniqueness of the solution*)
Theorem kantoro_unic:
∀ xs2:R, c_disc X0 (2*b0) xs2→ f xs2 = 0 → Un_cv Xn xs2.
```

To clarify the statements above, we mention that $c\_I\ a\ b$ and $c\_disc\ X0\ eps$ denote the closed interval $[a, b]$ and the closed disc centered in $X0$ and of radius esp, respectively. Hypothesis pr says that function f is derivable on interval $[a, b]$, hypothesis $Hder\_f$ states that f' is equal to the derivative of f on the same interval. The continuity and the bounding condition required for the derivative are formalized by $Hcont\_f'$ and $Hlim\_f'$.

The proof is done in classical logic, but we believe that all reasoning used can be made constructive. The development has around 1800 lines and can be found on Internet [1].

4. Multivariate analysis

The main difficulty was to generalize the work done for the real function case to several dimensions.

Mathematically, the elements of $\mathbb{R}^p$ are vectors of length p of real elements. Operations like addition of two vectors (+) and multiplication of a vector by a scalar (*) are defined component wise, using the operations on the real numbers so that $(\mathbb{R}^p, +, *)$ is a real vector space. One can further define various norms on these vectors. An important issue is working with functions $f : \mathbb{R}^p \to \mathbb{R}^p$ and being able to express concepts such as continuity and partial derivatives of such functions. An important amount of work concerns matrices, as some differentiation concepts are expressed using them (e.g. the Jacobian matrix of a function).

4.1. How to work in higher dimensions

4.1.1. Related work

The only proof assistant that contains a formalization of analysis in several dimensions is HOL Light. Among the concepts that can be found in the "Multivariate" library we can mention vectors, matrices, determinants, topology concepts for euclidean spaces, convexity, continuity, differentiability, integration. A documentation of this library can be found in [15]. It discusses the techniques that could be used for expressing concepts in multivariate calculus.

The strategy chosen is to implement vectors as functions $N \to real$, where N is a type with finite cardinality and $real$ is the type of real numbers in HOL. Vectors are of type $(N)finite\_image \to real$, where $(N)finite\_image$ has the same size as N when N is a finite type and 1 otherwise. An indexing operator is defined to use natural numbers as indexes.

Within this formalization, basic operations on vectors are easily defined. The representation of other definitions and properties is described. There are concepts from linear algebra: operators, matrices, determinants; topology: open, closed, compact, convex sets, sequences, continuity, differentiability; basic calculus theorems: mean value theorem, inverse function theorem.

4.1.2. Approaches for Coq

The main challenge is to find an appropriate representation for real vectors inside the Coq proof assistant. It is worth trying to find new solutions and not just copying the HOL version since Coq has a richer type system that supports dependent types and therefore offers more possibilities.

Taking into consideration some key issues of the concepts we need to formalize and the features of our theorem prover, we took a closer look at four approaches that seemed good candidates.

[1] http://www-sop.inria.fr/marelle/Ioana.Pasca/kantoro.v (the code works with version 8.1pl2 of Coq)

1. Elements of $\mathbb{R}^p$ can be viewed as functions of type $nat \to R$, that take as argument a natural number and return the real corresponding to that position in the vector. Then we say we take into consideration only the first p components (i.e. corresponding to the natural numbers smaller than p). Working with such an approach provides a rather natural way of handling the objects. For example, the norm is just a recursive function.
   The proof technique mostly used is induction on natural numbers.
   The disadvantage is that, in fact, our functions represent infinite vectors. It is difficult to state equality between such objects.
2. Think of vectors as a function $bound\ p \to R$, where $bound\ p$ is a type of dependent pairs formed from a natural number i and a proof that i is smaller than p. This representation raises problems even at early stages of development. For example, if one tries to define a function that takes as argument a vector and returns its maximum element, the first attempt would be to write it as a recursive function: the maximum between the i^{th} element and the maximum of the vector composed of the first $i-1$ elements. The problem here is that when talking about "vectors of $i-1$ elements" they are a different type than those with i elements and they must be constructed from "vectors of i elements" as must be the proof that the natural number passed as argument is smaller than $i-1$.
3. The third approach is to use the notions on vectors and matrices already developed by Nicolas Magaud [18], where the vectors are implemented as dependent lists. Therefore, $vect\ R\ p$ represents a list having p elements of type R.
   Recursion principles are defined for reasoning on this structure. Also, basic operations such as addition, multiplication by a scalar, scalar product are defined. Matrices are implemented as vectors of vectors and ring structure properties are proved. One of the reasons for which proofs are not easy in this implementation is because even basic functions, like vector addition, require complicated forms of dependent recursion.
4. Another approach is "borrowed" from a development based on Coq with the `ssreflect` extension and library [11, 12]. Among other things, it provides formalizations for various finite types and their properties. A finite type is a record containing a sort, a list of all elements of the finite type and a predicate saying that this list doesn't contain duplicates. We used the finite type $ordinal\ p$ which represents the set of natural numbers smaller than p. The implementation of finite types allows us to interpret this subset of natural numbers both as a type and as a list of elements. The real vectors have type $ordinal\ p \to R$.
   The most important facility offered by this approach is a dual vision of the vectors, more precisely, of the set that indexes the elements of a vector. It combines the views of the previous approaches. So we have a simple way of viewing the vectors as functions (as in the first or second approach), we can define their length (p) and we have a way of reasoning on the data structure: induction on the structure of the list (like in the third approach)enumerating the elements of $ordinal\ p$. Because vectors are just functions, one has easy access at their elements, just by providing the index and equality between two vectors can be stated by assuming extensionality.

Conclusion. We tested and compared the four approaches by defining the basic concepts in each of them. We formalized vectors, matrices, norms, functions and tried to do simple proofs around them so we would be able to understand the advantages and difficulties that occur. We took into consideration how easy it is to work with the concepts and how accurate they represent the mathematical objects and we decided to continue the formalization in the lines of the last approach proposed.

4.2. The formalized concepts

For our work in the multidimensional case we are in the following settings: Coq with the `ssreflect` extension and library, `Reals` library (which implies the use of classical logic), the extensionality

property and the axiom of choice. We have explained above our decision to use `ssreflect` and the libraries developed on it. The main reason for assuming general extensionality is to be able to state equality between two vectors or matrices. The classical logic settings are imposed by the use of the `Reals` library and it allows us to prove more results than in an intuitionistic setting. The axiom of choice has also proved necessary and we shall provide examples later on in this section to motivate its use.

The description of the mathematical concepts follows [20, 21, 7].

Before beginning to describe how concepts are represented, we would like to present in more detail the features of the library developed within the Mathematical Components project [11]. Their work offers descriptions for finite sets. We are interested in *ordinal p* which describes the set $\{0, 1, \ldots, p-1\}$. The type of the finite set *ordinal p* contains an enumeration of all the elements in the set. This enumeration is viewed as a list. As a consequence, we have a means for reasoning on such a data structure, i.e. induction on the structure of the list. Other features of this development will prove useful in our work as we will discuss in this section.

We begin by defining our set of indexes.

```
Definition I := ordinal p.
```

Under the hypothesis $0 < p$, the elements of $\mathbb{R}^p$ would be $I \to R$, i.e. functions from the set of indexes to the reals. This corresponds to the familiar way of viewing vectors as $(x_0, x_1, \ldots, x_{p-1})$, where $x_i \in \mathbb{R}, \forall i \in \{0, 1, \ldots, p-1\}$.

We define the operations on our vectors like addition, subtraction or multiplication by a scalar component wise.

```
Definition add_v (v1 v2:I→R):I→R := (fun i:I => v1 i + v2 i).
Definition dif_v (v1 v2:I→R):I→R := (fun i:I => v1 i - v2 i).
Definition mult_sv (a:R) (v:I→R):I→R := (fun i:I => a * v i).
```

Properties of these operations are proved by simply reducing them to properties on the real numbers. For the latter we benefit from tactics like `ring, field` and `fourier` provided by Coq which automatically solve a large variety of equalities and inequalities on the reals.

Here is a simple example that states the distributivity of scalar multiplication over addition:

```
Lemma mult_add:
∀ (a:R) (v1 v2 :I→R),
mult_sv a (add_v v1 v2) = add_v (mult_sv a v1) (mult_sv a v2).
Proof.
move=> a v1 v2.
rewrite -ext_eq /add_v /mult_sv =>i; ring.
Qed.
```

The tactics are part of the `ssreflect` extension. The first line of the proof just moves the variables a, v_1, v_2 from the goal to the context. The second line says that we want to prove our equality by resorting to the extensionality property (i.e. two vectors are equal iff each of their components for the same index are equal). After that, just rewriting the definition of addition and multiplication by a scalar is enough to obtain an equality between two real numbers. This latter equality is dealt with by Coq's tactic `ring`.

For the norm on vectors we chose the following:

$$\|x\| = \max_i |x_i|$$

to respect the conventions in [7]. Nevertheless, the structure of the proof does not depend on the particular norm used and it can be adapted to any other.

In formalizing the norm in Coq, we used once more the facilities provided by the `ssreflect` libraries. Within this development we can find the definition and proofs of various properties for folding a function over a finite set by using the generic function *iprod*. In our case, we fold the function *Rmax* (that gives the maximum of two real numbers) over our set of indexes I in order to retrieve the maximum value of a vector. The absolute value is easily computed component wise. In Coq, the definitions look like this:

```
Definition Rabs_v (v:I→R) (i:I):R := Rabs(v i).
Definition norm (v:I→R):R := iprod R Rmax 0 I (setA I) (Rabs_v v).
```

Proving the good properties for the norm like positive homogeneity, triangle inequality and positive definiteness is done by using properties already proven for *iprod* and induction on the structure of the list of indexes.

We can now define the distance corresponding to our norm:

```
Definition dist_Rp (u v:I→R) :R := norm (dif_v u v).
```

The properties for the distance follow naturally from those of the norm to ensure that, in our representation, $\mathbb{R}^p$, equipped with the above defined distance, is a metric space.

We now define the type of vector sequences as $nat \to I \to R$. The concepts of convergence and Cauchy criterion are formalized in the terms of our distance:

```
Definition conv_Rp (un:nat→I→R) (l:I→R) : Prop :=
   ∀ eps:R, 0 < eps→ ∃ N : nat,
      (∀ n:nat, (N ≤ n)%nat → dist_Rp (un n) l < eps).
Definition Cauchy_crit_Rp (vn:nat→I→R): Prop :=
   ∀ eps:R, 0 < eps → ∃ N : nat, (∀ n m:nat,
      (N ≤ n)%nat → (N ≤ m)%nat → dist_Rp (vn n) (vn m) < eps).
```

We proved a collection of results: the uniqueness of the limit of a sequence, any convergent sequence satisfies Cauchy's criterion, $\mathbb{R}^p$ is a complete metric space (i.e. any sequence satisfying Cauchy's criterion is convergent), the convergence in $\mathbb{R}^p$ is a convergence on components. This is, for example, a place where the axiom of choice turned out to be indispensable.

We then formalized functions $\mathbb{R}^p \to \mathbb{R}^p$ as $(I \to R) \to I \to R$. Operations on functions (addition, multiplication by a scalar etc.) are defined component wise, as we did for vectors.

Having the metric space structure enables us to express that the limit of a function f in a point $v0$ of this space is l:

```
Definition limit_Rp (f:(I→R)→I→R) (v0:I→R) (l:I→R):=
   ∀ eps:R, eps > 0→ ∃ alp:R, alp > 0 ∧
     (∀ v:I→R, 0 < dist_Rp v v0 < alp → dist_Rp (f v) l < eps).
```

To verify the feasibility of the approach we did some proofs around these concepts. For example: if there exists a limit of a function in a point then this limit is unique; the limit of the sum of two functions is the sum of the limits etc. We were rather pleased with the fact that our proofs could be done by following the "proof on paper" and the structure of their equivalent in the unidimensional case.

We can define functions continuous at a point by saying the limit at that point is the value of the function:

```
Definition cont_Rp (f:(I→R)→I→R) (v0:I→R):=
   limit_Rp f v0 (f v0).
```

In order to be able to consider partially derivable functions and their representation as the Jacobian matrix, we first have to introduce concepts on square matrices. They are of type $(I \to I \to R)$. Addition, subtraction and multiplication by a scalar are defined component wise. However, multiplication of a matrix and a vector and of two matrices are trickier to define. Having a square matrix of size p, $M = [m_{ij}]$ and a vector of size p, $v = (v_i)$, their product is $M * v = (\sum_{j=0}^{p-1} m_{ij} * v_j)$. We use a function to compute each term and another one to get the sum:

```
Definition mult1 (m:I→I→R) (v:I→R) (i j :I): R := m i j * v j.
Definition mult_mv (m:I→I→R) (v:I→R) (i:I): R :=
   iprod R Rplus 0 I (setA I) (mult1 m v i).
```

We define the multiplication of two matrices similarly.

Properties of operations on matrices are not difficult to verify once some needed results are obtained for the summation operator.

The norm on matrices is the following:

$$\|M\| = \max_i \sum_j |m_{ij}|$$

Among the properties we were able to prove is the following inequality

$$\|Mv\| \leq \|M\|\|v\|$$

where M is a matrix and v a vector.

```
Lemma ineq_norm_mv:
  ∀ m v, norm (mult_mv m v) ≤ norm_m m * norm v.
```

Other results like $\|M_1 M_2\| \leq \|M_1\|\|M_2\|$ were left unproven. We chose to concentrate on other parts and leave these for later, as it is our belief that they do not require more sophisticated techniques than those already tested.

The part of matrix theory where we encountered some difficulties was the one concerning the inverse of a matrix. The concepts "on paper" need some adaptation before being transposed on a computer. For example, we want to be able to talk about the inverse of a matrix even if we don't know yet whether it is invertible. But we should have the good properties only when we know the matrix is invertible. The solution is similar to the one for division in the `Reals` library and our proposal for derivatives in the one dimensional case. We define a total function on matrices that simulates our inverse, but which acts as the inverse only for the matrices satisfying the property.

Continuing our development with partial derivatives and differentiation, we have managed to formalize some rather important properties. For example, we proved that if a function is partially derivable, then the partial derivative with respect to the variable x_i can be interpreted as the derivative of a function with one argument.

Another useful result is: given a matrix valued function, $f : \mathbb{R}^p \to M_p(\mathbb{R})$, continuous at a point a, a sequence of vectors $\{x_n\}_{n\in\mathbb{N}}$ converging to a and a sequence $\{y_n\}_{n\in\mathbb{N}}$ converging to zero, then the product of $f(x_n)y_n$ will also be a sequence converging to zero.

The most important result of this field for our work is the proof of Taylor's formula for functions of class $C^{(2)}$. The statement and the proof are as follow:

Theorem 3 *Let $A \subset \mathbb{R}^p$ be a convex and open set. If $f : A \to \mathbb{R}$ is twice partially derivable with continuous first and second partial derivatives, then for all $a \in A$ and $v \in \mathbb{R}^p$ with $[a, a+v] \subset A$, there exists $c \in (a, a+v)$ so that $f(a+v) = f(a) + \sum_{i=1}^{p} \frac{\partial f(a)}{\partial x_i} v_i + \frac{1}{2!} \sum_{i,j=1}^{p} \frac{\partial^2 f(c)}{\partial x_i \partial x_j} v_i v_j$.*

Proof. Consider

$$g : [0,1] \to \mathbb{R},\ g(t) = f(a + tv)$$

then g is twice derivable on $[0,1]$ and

$$g'(t) = \sum_{i=1}^{p} \frac{\partial f(a+tv)}{\partial x_i} v_i \tag{6}$$

$$g''(t) = \sum_{i,j=1}^{p} \frac{\partial^2 f(a+tv)}{\partial x_i \partial x_j} v_i v_j \tag{7}$$

From the Taylor formula in one dimension we get that there exists $\eta \in (0,1)$ so that

$$g(1) = g(0) + g'(0) + \frac{1}{2!} g''(\eta)$$

which gives us the desired result for $c = a + t\eta \in (a, a+v)$.

The proof of this theorem is based on the proof of the Taylor formula in one dimension, which we also formalized. Also, an important issue for this proof is to show some relations between various concepts of differentiability, i.e. to prove equalities (6) and (7).

The development is at this point concentrating on Taylor's formula. Once this result is proved, the only pieces still missing for a complete formalization of the theorem, will be some results from matrix theory. The choice not to focus on them was made because formalization of matrix theory is on-going work for the `ssreflect` libraries and most of the results we need should be available in the near future.

5. Formal proof in the multidimensional case

The proof for the multidimensional case follows the same structure described in Section 2.3. Also, the formalization done for the real case was a good guide in our development. Some results generalize nicely from one to several dimensions. For example, the properties of the absolute value function naturally generalize to those of the norm.

Other results, however, do need considerably more work than their real counterparts. Take for example the proof that

$$|ax| = |a||x| \ \ \forall a, x \in \mathbb{R}$$

This is a trivial property of the absolute value function.

The equivalent result we need in the multidimensional case is:

$$\|Ax\| \leq \|A\|\|x\| \ \forall A \in M_{p\times p}, x \in \mathbb{R}^p$$

The proof of this property, however, is far from trivial. On paper the proof mainly deals with inequalities on sums and absolute values. We managed to obtain a formal proof that follows the same line of reasoning. The technique used is induction on the list enumerating the indexes of a vector (as vectors are represented as functions from the list of indexes to R). Intuitively this corresponds to an induction on the dimension of $\mathbb{R}^p$.

Another example of how things get really complicated in several dimensions is the following:

From the inequality $|1 - t| \leq \frac{1}{2}$ one can easily infer that $t \neq 0$.

But having the relation $\|E_p - A\| \leq \frac{1}{2}$, where E_p is the unit matrix of size p, does not trivially imply that A is invertible. In fact, the proof is rather complicated even on paper as it uses results

from the theory of functional spaces, like the convergence of a particular type of matrix series. This is among the results on matrices we have not formalized yet.

In this setting, we provide a verification of the theorem structure. We were pleasantly surprised by the resemblance between this proof and the one we had already done for the real case. For secondary results we mostly needed a generalization of those in one dimension. Also, the structure of the proof for the main theorem is basically the same.

Provided the appropriate declarations of variables and hypothesis, the formal statement of the existence theorem is the following:

```
Theorem kantoroRp_exist:
∃ xs:I→R,
conv_Rp Xn xs ∧ norm (dif_v xs X0) ≤ 2*b0 ∧ f xs = vect0.
```

The development so far has more than 3000 lines of code [2].

6. Conclusions

The degree of difficulty in providing a formal proof for a theorem depends heavily on how well the basic concepts involved are described. It is important to have concepts that are easy to manipulate and correspond to the intuition about the mathematical object they model. This way, formal theorem proving comes rather naturally to one that is used to doing a proof "on paper". For example, having a new approach to derivatives, a new way to handle them, made our work much easier.

Trying to find an adequate formalization for real vectors revealed once more the difference made by a good representation and good proof techniques. We are rather proud of our choice, as it enabled us to obtain quite a large amount of results in a relatively short time.

The work in formalizing multivariate analysis concepts is new for Coq. Though far from offering an extensive coverage of the properties needed, it is a first step towards having such a library.

6.1. Future work

Formal study of Newton's method and Kantorovitch's theorem is important for computer science domains where solving equation systems occurs frequently. Having a formal description of this theorem should make it easier to study whether better convergence criteria can be found when the studied function satisfies special properties. Working with special cases of input functions is often tedious, and having a formalized proof ensures that no special condition will be overlooked. This means that on one hand, the algorithm that does the computation is certified and on the other, special cases are automatically treated in order to obtain better performances from these algorithms.

There is a lot of work that still remains to be done in order to attain such a goal. For the moment, the work accomplished offers an insight on the difficulties that occur in such a development, proposes some potential solutions and gives the formalization of some basic results.

For future work, in the short term, there are still some results that need to be verified in order to have a complete formal proof for the multidimensional case. Then we will be able to concentrate on special cases of the theorem in order to improve efficiency of algorithms. In an even longer term, we would like to study and develop techniques and tools that will enable us to address numerical analysis problems from the formal proof point of view.

[2] http://www-sop.inria.fr/marelle/Ioana.Pasca/multidim/

7. Acknowledgments

This work is based on the author's Master thesis. The author thanks Yves Bertot for his supervision, advice and support.

References

[1] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development, Coq'Art:the Calculus of Inductive Constructions*. Springer-Verlag, 2004.

[2] Coq development team. *The Coq Proof Assistant Reference Manual, version 8.1*, 2006.

[3] Luís Cruz-Filipe. A Constructive Formalization of the Fundamental Theorem of Calculus. In H. Geuvers and F. Wiedijk, editors, *Types for Proofs and Programs*, volume 2464 of *LNCS*, pages 108–126. Springer-Verlag, 2003.

[4] Luís Cruz-Filipe, Herman Geuvers, and Freek Wiedijk. C-CoRN, the Constructive Coq Repository at Nijmegen. In *MKM*, pages 88–103, 2004.

[5] D. Delahaye and M. Mayero. Dealing with algebraic expressions over a field in Coq using Maple *Journal of Symbolic Compuations*, 39(5):569-592, May 2005.

[6] B. Duerte. Elements of Mathematical Analysis in PVS. In *Proceedings of the Ninth International Conference on Theorem Proving in Higher-Order Logics (TPHOL '96)*, 1996.

[7] B. Démidovitch et I. Maron. *Éléments de Calcul Numérique*. Mir - Moscou, 1979.

[8] Jacques D. Fleuriot. On the Mechanization of Real Analysis in Isabelle/HOL. In J. Harrison and M. Aagaard, editors, *Theorem Proving in Higher Order Logics: 13th International Conference, TPHOLs 2000*, volume 1869 of *Lecture Notes in Computer Science*, pages 146–162. Springer-Verlag, 2000.

[9] R. Gamboa and M. Kaufmann. Nonstandard Analysis in ACL2. *Journal of automated reasoning*, 27(4):323–428, November 2001.

[10] H. Geuvers, F. Wiedijk, and J. Zwanenburg. A Constructive Proof of the Fundamental Theorem of Algebra without Using the Rationals. In P. Callaghan, Z. Luo, and R. Pollack, editors, *Types for Proofs and Programs, Proc. of the International Workshop TYPES 2000*, volume 2277 of *LNCS*, pages 96–111. Springer, 2001.

[11] Georges Gonthier. Notation of the Four Colour Theorem Proof. Available at http://research.microsoft.com/ gonthier/4colnotations.pdf.

[12] Georges Gonthier, Assia Mahboubi, Laurence Rideau, Enrico Tassi, and Laurent Théry. *A Modular Formalisation of Finite Group Theory*, Rapport de Recherche 6156, INRIA, 2007.

[13] Michael J. C. Gordon and Thomas F. Melham. *Introduction to HOL : A Theorem Proving Environment for Higher-order Logic*. Cambridge University Press, 1993.

[14] John Harrison. *Theorem Proving with the Real Numbers*. Springer-Verlag, 1998.

[15] John Harrison. A HOL Theory of Euclidian Space. In Joe Hurd and Thomas F. Melham, editors, *TPHOLs*, volume 3603 of *LNCS*, pages 114–129. Springer, 2005.

[16] J. Harrison and L. Théry. A Skeptic's Approach to Combining HOL and Maple. *Journal of Automated Reasoning*, 21(3):279-294, December 1998.

[17] Matt Kaufmann, Panagiotis Manolios, and J. Strother Moore. *Computer-aided Reasoning: an Approach*. Kluwer Academic Publishing, 2000.

[18] Nicolas Magaud. *Programming with Dependent Types in Coq: A Study of Square Matrices*, Coq contribution: http://coq.inria.fr/contribs-eng.html.

[19] Micaela Mayero. *Formalisation et Automatisation de Preuves en Analyses Reelle et Numerique.* PhD thesis, Université de Paris VI, 2001.

[20] Mihail Megan. *Analiza Matematica.* Mirton Timisoara, vol. 1-2, 1999.

[21] Mihail Megan. *Calcul Diferential si Integral in Rp.* Mirton Timisoara, 2000.

[22] Milad Niqui. *Formalising Exact Arithmetic: Representations, Algorithms and Proofs.* PhD thesis, Radboud University, Nijmegen, September 2004.

[23] Lawrence C. Paulson and Tobias Nipkow. *Isabelle : A Generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science.* Springer-Verlag, 1994.

[24] A. Robinson. *Non-Standard Analysis.* Princeton University Press, 1996.

[25] Natarajan Shankar, Sam Owre, and John M. Rushby. *The PVS Proof Checker: A Reference Manual.* Computer Science Laboratory, SRI International, Menlo Park, CA, February 1993. A new edition for PVS Version 2 is expected in 1998.

Vérification formelle d'un algorithme d'allocation de registres par coloration de graphe

Sandrine Blazy & Benoît Robillard & Éric Soutif

Laboratoire CEDRIC, 292 rue Saint-Martin, 75141 Paris CEDEX 03
{blazy,robillard}@ensiie.fr,soutif@cnam.fr

Résumé

Le travail présenté dans cet article est à l'interface entre la recherche opérationnelle et les méthodes formelles. Il s'inscrit dans le cadre du projet CompCert ayant pour but le développement et la vérification formelle, utilisant l'assistant de preuve Coq, d'un compilateur du langage C potentiellement utilisable pour la production de logiciels embarqués critiques. Nous nous intéressons dans cet article à l'allocation de registres, qui consiste à optimiser l'utilisation des registres du processeur. Nous proposons d'aborder cette optimisation en la modélisant par un problème dit de coloration avec préférences dont nous vérifions formellement la résolution. Cette vérification prend deux formes : preuve de correction de la spécification Coq pour la première partie de l'algorithme et validation *a posteriori* pour la seconde.

Introduction

Les méthodes de développement formelles permettent de produire du code source certifié. Cependant, une faille du processus de certification demeure au niveau de la compilation de ce code. En effet, un bug dans le compilateur peut introduire des erreurs dans le code assembleur généré, et donc invalider le code source. De là sont nés le besoin de compilateurs certifiés et le projet CompCert. Ce projet a pour but de développer avec l'assistant à la preuve Coq un compilateur réaliste d'un vaste sous-ensemble du langage C vers le langage assembleur du processeur PowerPC, principalement dévolu au domaine des logiciels embarqués [Ler06, BDL06].

Le compilateur CompCert est un compilateur modérément optimisant, qui accomplit de nombreuses passes de transformations de programmes. Celui-ci est certifié, c'est-à-dire qu'il est accompagné d'une preuve Coq de préservation du comportement des programmes (tout au long du processus de compilation). Cette preuve consiste à établir la préservation sémantique de chaque passe du compilateur. Il s'agit d'écrire en Coq une passe de compilation et de prouver ensuite que celle-ci transforme un programme en un programme observationellement équivalent. L'intérêt d'une telle approche est que le compilateur est certifié une fois pour toutes, indépendamment des programmes à compiler.

Le développement du compilateur CompCert est une expérience de conception assistée par preuve. Il ne s'agit pas de prouver un compilateur existant, mais plutôt de définir conjointement les langages intermédiaires, les transformations de programmes et les preuves associées. C'est souvent à l'issue d'une preuve jugée trop difficile qu'il est décidé de modifier un langage intermédiaire, voire de définir un nouveau langage intermédiaire, ce qui nécessite de plus de prouver à nouveau certaines propriétés

ayant préalablement été prouvées. Ainsi, le compilateur CompCert actuel dispose de sept langages intermédiaires.

L'allocation de registres est la seule phase du compilateur CompCert qui n'est pas entièrement écrite en Coq. La raison principale est qu'il s'agit d'une transformation de programmes peu adaptée à une écriture fonctionnelle, et que l'effort nécessaire pour spécifier en Coq et prouver ensuite la préservation sémantique est trop important. En effet, classiquement, l'allocation de registres se ramène à un problème de coloration avec préférences de graphe. Ce problème étant $\mathcal{NP}$-difficile, CompCert implante une des heuristiques les plus performantes pour le résoudre.

Dans CompCert, l'allocation de registres est écrite en Caml puis validée *a posteriori* en Coq : pour chaque programme à compiler, il est vérifié formellement que la solution calculée par l'allocation de registres est correcte. L'intérêt de cette approche est que la preuve à effectuer est beaucoup plus facile, puisqu'il s'agit de vérifier que l'allocation de registres a bien renvoyé une coloration du graphe. En outre, cette technique permet de valider une transformation de programme qui n'a pas été écrite en Coq.

Cet article décrit une nouvelle méthode d'optimisation de l'allocation de registres pour le compilateur CompCert. Nous détaillons d'abord une formalisation en Coq d'un algorithme de coloration (sans préférences) adapté à une famille de graphes regroupant la majeure partie des graphes utilisés pour modéliser l'allocation de registres. Il s'agit d'un algorithme exact, dont nous prouvons également l'optimalité en Coq dans la famille de graphes précitée. Nous présentons ensuite la seconde partie de la méthode qui utilise la programmation linéaire en nombres entiers. Cette étape est réalisée par un solveur qui fournit une allocation de registres optimale. Comme il s'agit d'un solveur externe, cette solution est validée *a posteriori* en Coq. Le programme mathématique à résoudre dépend du résultat de la première partie de la méthode, d'où l'importance de l'optimalité du premier algorithme.

L'objectif de cet article est double. D'une part, nous proposons une amélioration de l'allocation de registres actuellement utilisée dans le compilateur certifié CompCert. D'autre part, nous présentons le début d'un travail de formalisation en Coq de structures de données et algorithmes de théorie des graphes et de programmation mathématique. Cet article est organisé comme suit. La première partie introduit l'allocation de registres et présente un état de l'art. Puis, la deuxième partie décrit les notions de théorie des graphes et de programmation mathématique utiles pour l'allocation de registres. Ensuite, la troisième partie explique notre algorithme d'allocation de registres. Enfin, la quatrième partie détaille la spécification Coq ainsi que les principales propriétés que nous avons prouvées. Le code source complet de ce développement est disponible sur la page `http://www.ensiie.fr/~blazy/register-allocation`.

1. Allocation de registres

1.1. Généralités

Au cours de l'exécution d'un programme, le processeur effectue un grand nombre d'accès à la mémoire afin de lire et écrire les valeurs des variables du programme. Ces accès sont naturellement gourmands en temps. Pour accélérer l'exécution des programmes, le processeur est muni d'un petit nombre de zones de stockage à accès beaucoup plus rapide, les registres. En général, le nombre de registres d'un processeur est nettement inférieur au nombre de variables utilisées dans un programme. Le but de l'allocation de registres est de déterminer où sont stockées les variables d'un programme à tout moment de son exécution : soit en registres si ces derniers sont disponibles, soit en mémoire le cas échéant. La difficulté est de proposer une affectation optimale des registres. Il est par exemple souvent nécessaire de choisir entre conserver une variable v dans un même registre R pendant l'exécution complète d'un programme (ce qui rend R inutilisable pour stocker d'autres variables), et réutiliser R lorsque la valeur de v n'a plus besoin d'être conservée en vue d'utilisations futures (ce qui nécessite

de transférer en mémoire la valeur de v).

L'allocation de registres est la passe de compilation la plus étudiée et la plus difficile à mettre en œuvre dans un compilateur. La qualité du code compilé dépend en effet de la qualité de l'allocation de registres. Les deux tâches principales de l'allocation de registres sont le *vidage* de registres en mémoire, et la *fusion* de registres. Le vidage décide quelles variables seront stockées ultérieurement en registres. La fusion tient compte le plus possible des préférences entre variables, afin de minimiser les transferts entre registres. Par exemple, une affectation $x = y$ entraîne une préférence entre les variables x et y correspondant à la condition optimisante, qu'au vu de cette affectation, il serait préférable de stocker x et y dans le même registre.

1.2. Approches heuristiques

L'allocation de registres consiste à minimiser le nombre d'accès à la mémoire, étant donné un nombre fixe de registres. Classiquement, ce problème se ramène à la recherche d'une coloration de graphe. Dans le cas de l'allocation de registres, il s'agit d'un graphe d'interférences (défini dans la section 2.1), obtenu suite à une analyse de vivacité du programme à compiler. Le problème de coloration étant dans le cas général $\mathcal{NP}$-difficile, la quasi-totalité des approches imaginées ont été heuristiques ([Cha82], [BCT94], [GA96], [PP05], ...). Ces heuristiques proposent différentes combinaisons des deux phases de vidage et de fusion. Les plus simples effectuent les deux phases de manière séquentielle tandis que d'autres, plus sophistiquées, les réalisent simultanément. Les heuristiques d'allocation de registres évoluent aujourd'hui encore, par exemple afin de tenir compte de la rapidité croissante des processeurs ainsi que du coût croissant des accès à la mémoire. Pour un compilateur d'un langage tel que C, l'heuristique la plus efficace à l'heure actuelle est celle d'Appel et George [GA96]. C'est d'ailleurs celle qui a été initialement choisie dans le compilateur CompCert.

Une autre heuristique récente est celle imaginée par Palsberg et Pereira [PP05]. Si son efficacité n'est pas suffisante pour remplacer l'heuristique d'Appel et George, ses fondements sont par contre forts intéressants. En effet, suivant la piste ouverte par Andersson [And03], ceux-ci ont mesuré qu'une large majorité des graphes d'interférences ont la propriété d'être triangulés [1]. Ils affirment en effet que plus de 95% des graphes d'interférences des méthodes de la bibliothèque Java 1.5 et des 27921 graphes de référence publiés par Appel et George ([AG05]) ont cette propriété.

L'allocation de registres est également une transformation de programmes qui :
- renomme les variables du programme,
- insère des instructions de lecture et écriture en mémoire, pour chaque variable à vider en mémoire,
- supprime des affectations lorsqu'elles concernent des variables stockées dans des registres ayant été fusionnés.

Il est donc important de s'assurer que l'allocation de registres préserve le comportement des programmes. De nombreuses approches de validation reposent sur l'utilisation de techniques d'analyse statique. Peu de travaux portent sur la vérification formelle (c'est-à-dire à l'aide d'un assistant à la preuve). [Oho04] propose un système de types dédié à l'allocation de registres, ainsi qu'un algorithme d'allocation de registres correct par construction, mais les instructions considérées sont celles d'un petit langage, et cette démarche semble difficilement applicable pour un compilateur tel que CompCert. [NPP07] propose un langage dédié à l'allocation de registres, ainsi qu'un système de types. La sûreté du typage de ce système de types a récemment été prouvée en Twelf. Le but de ce travail est de fournir un cadre général permettant de comparer différentes stratégies d'allocation de registres.

[1] La définition d'un graphe triangulé est donnée dans la section 2.2.

1.3. Programmation linéaire en nombres entiers appliquée à l'allocation de registres

Les premiers travaux utilisant la programmation linéaire en nombres entiers pour l'allocation de registres sur des problèmes de petite taille furent ceux de Goodwin et Wilken en 1996 [GW96] sur architecture CISC. Les résultats furent encourageants mais pas suffisamment pour supplanter l'approche traditionnelle. Il s'agissait alors d'une formulation incluant phases de vidage et de fusion. En 2001, Appel et George [AG01] ont introduit une formulation de la phase de vidage pour les processeurs à architecture CISC[2] puis ont appliqué une méthode heuristique pour la phase de fusion (leurs tentatives d'utilisation de la programmation mathématique pour la phase de fusion se sont avérées infructueuses, particulièrement en raison du temps de résolution du problème par le solveur). Les résultats furent meilleurs que ceux de Goodwin et Wilken.

Cette année, Grund et Hack [GH07] ont élaboré un algorithme de coupes (*i.e.* une extension de la résolution par programmation mathématique) pour obtenir un résultat optimal pour la phase de fusion. Leur démarche a permis d'obtenir un résultat optimal pour 471 des 474 graphes de l'*Optimal Coalescing Challenge*[3] dans des temps très raisonnables pour la plupart des cas (430 cas sont traités en moins de 6 secondes).

2. Fondements mathématiques

2.1. Modélisation graphique de l'allocation de registres

Une *coloration* d'un graphe G est une fonction qui à chaque sommet de G associe une couleur de sorte que pour toute arête (i, j) de G les couleurs associées à i et j sont différentes. Si p est un entier strictement positif, une coloration utilisant moins de p couleurs est appelée *p-coloration*. Une coloration est dite *partielle* si certains sommets ne sont pas colorés. Une coloration est dite *optimale* si elle utilise un nombre minimal de couleurs.

Dans le cas d'une allocation de registres, le graphe à colorier est un *graphe d'interférences*, dont les sommets représentent les variables du programme à compiler. Les arêtes sont de deux types. Les arêtes d'*interférence* relient tous les sommets qui représentent des variables qui ne doivent pas occuper les mêmes registres à un instant donné de l'exécution du programme. Les arêtes de *préférence* relient tous les sommets qui représentent des variables telles qu'il existe une instruction d'affectation entre celles-ci (et il n'existe pas d'arête d'interférence entre ces sommets). Un poids est associé à chaque arête afin de tenir compte de la fréquence d'exécution des instructions, ainsi que de la fréquence d'utilisation des variables.

Dans ce qui suit, nous définissons un graphe G comme étant un triplet (S, I, P), où G est le graphe dont l'ensemble des sommets est S, l'ensemble des arêtes d'interférence est I et l'ensemble des arêtes de préférences est P. De plus, les graphes formés par S et I d'une part et S et P d'autre part sont respectivement appelés *interf-graphe* et *pref-graphe* de G.

Soient un entier strictement positif p et un graphe G. Le problème de *p-coloration avec préférences* consiste à trouver une p-coloration partielle de l'interf-graphe de G qui minimise la fonction $f = \sum_{(i,j)\in D} w_{ij} + c|NC|$, où D est l'ensemble des arêtes de préférences dont les extrémités sont de couleurs différentes, NC est l'ensemble des sommets non colorés, w_{ij} est le poids associé à l'arête (i, j), et c est une constante représentant le coût de vidage en mémoire d'un sommet non coloré. Une p-coloration avec préférences est optimale si elle minimise la fonction f.

[2] La particularité de l'architecture CISC est que lors d'une opération du processeur, certains opérandes peuvent être en mémoire, d'autres en registres, contrairement à l'architecture RISC dans laquelle tous les opérandes doivent être en registre.

[3] Il s'agit d'une bibliothèque de graphes de référence publiée par Appel pour l'optimisation de la phase de fusion.

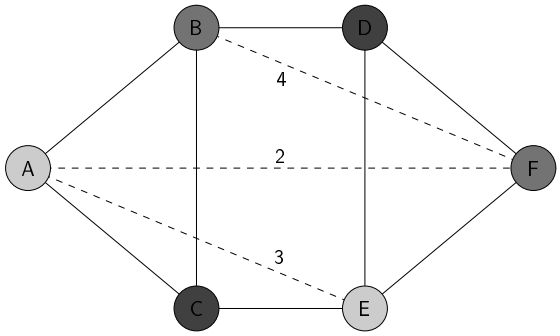


FIG. 1 – Instance du problème de 3-coloration avec préférences.

L'allocation de registres est finalement modélisée par le problème de k-coloration avec préférences appliqué au graphe d'interférences, où k désigne le nombre de registres utilisables. Chaque couleur représente un registre. La figure 1 est un exemple d'instance résolue du problème de 3-coloration avec préférences. En trait plein sont représentées les arêtes d'interférence et en pointillé les arêtes de préférences. Dans cet exemple la coloration réalisée est optimale puisqu'elle colore tous les sommets et que deux des trois arêtes de préférence ont des couleurs identiques aux deux extrémités. En effet, il est impossible de satisfaire les trois préférences car sinon les sommets A, B, E et F seraient de la même couleur et donc plusieurs contraintes de coloration seraient violées.

Étant donnée cette modélisation, la phase de vidage des registres en mémoire consiste à rechercher un ensemble de sommets k-colorable, c'est-à-dire pouvant être coloré avec k couleurs. La phase de fusion consiste à colorer cet ensemble de sommets.

2.2. Théorie des graphes et programmation mathématique

Les travaux décrits dans cet article nécessitent la définition de notions de théorie des graphes et de programmation mathématique. Nous commençons par présenter la classe des graphes triangulés qui joue un rôle central au sein de notre étude ainsi que les ordres d'élimination simpliciaux qui leur sont intimement liés pour finir par une description de la programmation linéaire en nombres entiers. Dans les définitions suivantes, $G = (S, I, P)$ désigne un graphe, et E désigne un ensemble de sommets de S.

1. Un cycle C est dit *sans corde* si aucune arête ne relie deux sommets non consécutifs de C.
2. Le *graphe induit* par E est la restriction de G aux sommets appartenant à E et aux arêtes reliant deux sommets de E.
3. Un graphe qui ne possède aucun cycle induit sans corde de longueur supérieure ou égale à quatre est dit *triangulé* (*chordal* en anglais).
4. Une *clique* est un graphe dont tous les sommets sont reliés deux à deux.
5. Un sommet s est *simplicial* dans un graphe G si le graphe induit par les voisins de s est une clique.
6. Un *ordre d'élimination simplicial* (noté *oes* ou *peo* pour *perfect elimination order* en anglais) est une permutation $(x_1, \ldots, x_n)$ de S telle que pour tout $i \in \{1, \ldots n\}, x_i$ est simplicial dans le graphe induit par $\{x_i, \ldots x_n\}$.

7. $n(G)$ désigne *l'ordre* de G, c'est-à-dire le nombre de sommets de G.
8. $m(G)$ est la *taille* de G, c'est-à-dire le nombre d'arêtes de G.
9. $\chi(G)$ est le *nombre chromatique* de G, c'est-à-dire le nombre minimal de couleurs nécessaire pour réaliser une coloration de G.
10. $\omega(G)$ est l'ordre de la plus grande clique induite de G.

Une autre méthode utilisée pour la résolution exacte de problèmes d'optimisation est la programmation mathématique. Il s'agit de décrire le problème comme la minimisation (ou la maximisation) d'une fonction (appelée *fonction économique*) sous contraintes de ses variables (égalités et inégalités). Tout programme mathématique (P) peut donc s'écrire sous la forme suivante, où X désigne l'ensemble des solutions admissibles et n désigne le nombre de variables :

$$(P) \begin{cases} Min & f(x) \\ & x \in X \subseteq \mathbb{R}^n \end{cases}$$

Nous parlerons particulièrement de la programmation linéaire en variables {0,1} (PLNE) c'est-à-dire où les variables appartiennent toutes à l'ensemble {0,1}, et où la fonction économique et toutes les contraintes du problème sont linéaires. Notons qu'il s'agit généralement de la résolution exacte de problèmes $\mathcal{NP}$-difficiles par des méthodes énumératives ce qui demande un temps de résolution exponentiel. La résolution est confiée à un solveur commercial (par exemple CPLEX [Ilo02]). L'intérêt de tels solveurs est le recours à des techniques sophistiquées qui permettent de diminuer le temps de résolution du programme mathématique.

3. Description de l'algorithme

3.1. Séparation des phases et non optimalité

L'approche qu'il a été décidé de suivre pour cette étude est analogue à celle suivie par Appel et George [AG01], c'est-à-dire la voie de la programmation linéaire en nombres entiers et du traitement séquentiel des phases de vidage et de fusion. Il s'agit cependant d'adapter la modélisation, car l'architecture du PowerPC (le langage cible de CompCert) est RISC, et non pas CISC comme dans [AG01]. Concrètement, les contraintes portant sur les variables des programmes linéaires ne sont pas les mêmes. Par exemple, l'architecture RISC oblige toute variable à être en registre au moment de son utilisation tandis que ce n'est pas obligatoire pour une architecture CISC. En contrepartie, un processeur à architecture RISC possède plus de registres dédiés au stockage des variables. Enfin, les instructions à considérer sont plus simples dans le cas d'un processeur RISC, et le nombre de contraintes est donc plus petit que dans le cas d'une architecture CISC.

Par ailleurs, contrairement à [AG01], nous traitons également la phase de fusion par programmation linéaire en nombres entiers. Ce traitement séquentiel pose un problème non négligeable. En effet, la programmation mathématique permet d'obtenir un résultat optimal pour la phase de vidage puis pour la phase de fusion mais ces deux optimisations sont dépendantes l'une de l'autre et peuvent donc ne pas déboucher sur une allocation des registres qui soit globalement optimale.

3.2. Deux processus de résolution

Nous pouvons néanmoins, dans la majorité des cas, éviter ce problème en limitant l'optimisation à la phase de fusion. En effet, la phase de vidage n'est utile que si le graphe d'interférences n'est pas k-colorable. Or, le grand nombre de registres allouables sur architecture RISC rend cette éventualité peu probable. Il suffit donc de tester si le graphe d'interférences est k-colorable pour savoir s'il est nécessaire d'effectuer la phase de vidage. Ce test est possible [4] notamment dans les graphes dont l'interf-graphe est

[4] En un temps raisonnable, où le test de k-coloration est dit polynomial.

triangulé, puisqu'il existe des algorithmes efficaces réalisant des colorations optimales dans ces graphes. Nous appliquons donc au graphe d'interférences un algorithme de coloration, l'algorithme de coloration gourmande, qui renvoie une coloration optimale si le graphe est triangulé et quelconque sinon. Si le nombre de couleurs utilisé par cette coloration est inférieur ou égal à k, il n'est pas nécessaire de réaliser la phase de vidage et donc le résultat de la phase de fusion correspond à une solution optimale du problème global d'allocation puisque cette dernière phase est traitée par programmation linéaire en nombres entiers.

La figure 2 résume cette approche. La vérification formelle en Coq de l'algorithme est composée de deux parties. La coloration est spécifiée et prouvée en Coq, tandis que les phases de vidage et de fusion sont validées *a posteriori*. Plus précisément, il est vérifié en Coq que les résultats calculés par le solveur externe représentent bien une coloration du graphe d'interférences.

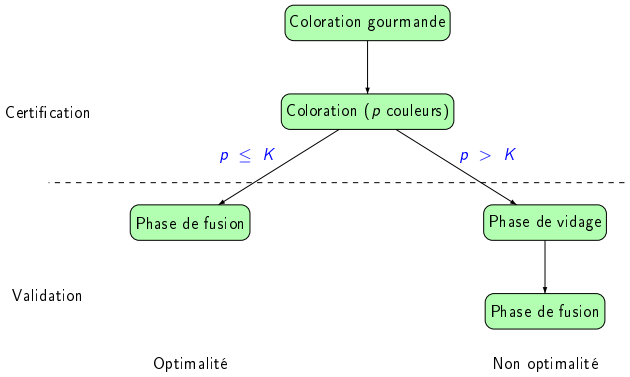


FIG. 2 – Principales étapes de l'algorithme.

3.3. Algorithme de coloration gourmande

Le test de k-colorabilité est effectué par l'algorithme de coloration gourmande. Cet algorithme fournit une coloration optimale (*i.e.* utilisant un nombre minimal de couleurs) si l'ordre dans lequel sont coloriés les sommets est l'ordre inverse d'un ordre d'élimination simplicial [Gav72]. Ces résultats ont été prouvés en Coq (*cf.* section 4.5). Il en découle que les graphes triangulés peuvent être colorés de façon optimale grâce à cet algorithme puisque la recherche d'ordre d'élimination simplicial est un problème pour lequel il existe divers algorithmes polynomiaux. Nous avons choisi d'écrire en Coq un algorithme de recherche d'un ordre d'élimination simplicial relativement naïf mais dont la correction est plus simple à montrer que pour les algorithmes les plus efficaces et dont la complexité (en temps) est comparable. L'algorithme est donné ci-dessous. Il consiste informellement à chercher un sommet simplicial s, le retirer du graphe et itérer ce procédé. Cette technique fonctionne car tout graphe induit d'un graphe triangulé est lui même triangulé et possède donc un ordre d'élimination simplicial.

L'appel à la fonction $is\_sv(s, S-T)$ teste si s est un sommet simplicial dans le graphe induit par $S-T$. Plus précisément, il est testé si s et ses voisins de $S-T$ forment une clique. De plus, un graphe est triangulé si et seulement s'il admet un ordre d'élimination simplicial [FG65]. Aussi, l'ordre renvoyé par l'algorithme 1 est un ordre d'élimination simplicial si et seulement si le graphe G est triangulé.

Algorithme 1 peo_search (G)

Entrée: Un graphe G=(S, I, P) dont l'interf-graphe est triangulé
Sortie: Un ordre $x = \{x_1, x_2, \ldots x_{n(G)}\}$ d'élimination simplicial de l'interf-graphe

1: $i := 0, U := \emptyset$
2: **tant que** $i < n(G)$ **faire**
3: trouvé := faux, T := U
4: **tant que** trouvé = faux **faire**
5: choisir s dans $S - T$
6: **si** $is\_sv(s, S - T)$ **alors**
7: $x_{i+1} := s$
8: trouvé := vrai
9: **sinon**
10: $T := T \cup s$
11: **fin si**
12: **fin tant que**
13: $i := i + 1; U := U \cup \{s\}$
14: **fin tant que**

Étant donné un ordre des sommets, l'algorithme 2 de coloration gourmande consiste à colorer les sommets selon cet ordre en affectant à chaque fois la plus petite couleur qui n'est utilisée par aucun des voisins du sommet courant x_i, en commençant la numérotation (coloration) à 1.

Algorithme 2 graph_coloring (G)

Entrée: Un graphe G
Sortie: Une coloration de G, optimale si G est triangulé

1: $x = peo\_search(G), U := \emptyset$
2: **pour tout** i de n(G) à 1 **faire**
3: T := $get\_nghbs(G, x_i), U := U \cup x_i$
4: affecter à x_i la plus petite couleur qui n'est affectée à aucun sommet de $T \cap U$
5: **fin pour**

3.4. Programmation linéaire en nombres entiers

Nous utilisons pour modéliser la phase de fusion sur un graphe $G = (S, I, P)$ le programme mathématique défini dans [GH07]. Il existe deux types de variables pour modéliser le problème : d'une part, les variables x_{ic} qui valent 1 si et seulement si le sommet i est de couleur c ; d'autre part les variables y_{ij} qui valent 1 si et seulement si (i, j) est une arête et i et j sont de couleurs différentes.

Le programme mathématique est défini dans la figure 3. Il comprend trois séries de contraintes :
- (C_1) à chaque sommet doit être affectée une et une seule couleur,
- (C_2) chaque arête d'interférence doit avoir des extrémités de couleurs différentes,
- (C_3) $y_{i,j}$ doit valoir 1 si i et j sont de couleurs différentes. En effet, si les couleurs sont différentes alors le membre droit de l'inégalité (C_3) vaut 1 lorsque c est la couleur de i.

Pour optimiser la coloration il suffit de minimiser le poids des arêtes de préférence dont les extrémités sont de couleurs différentes, c'est-à-dire à minimiser $f = \sum_{(i,j)\in P} w_{ij} \times y_{ij}$.

Nous avons également défini un modèle mathématique adapté au traitement simultané des deux phases. Ce modèle étant assez proche du précédent, il n'est pas présenté dans cet article. Il suffit en effet d'ajouter des variables x_{i0} qui valent 1 si et seulement si le sommet i n'est pas coloré et de

$$
(P1)\begin{cases}
Min & \displaystyle\sum_{(i,j)\in P} w_{ij} \times y_{ij} & & \\
\text{sous les contraintes} & & & \\
(C_1)\ \forall i\ \in \{1,\ldots,n(G)\}, & \displaystyle\sum_{c=1}^{k} x_{ic} & = & 1 \\
(C_2)\ \forall (i,j) \in I,\ \forall c \in \{1,\ldots,k\}, & x_{ic} + x_{jc} & \leq & 1 \\
(C_3)\ \forall (i,j) \in P,\ \forall c \in \{1,\ldots,k\}, & x_{ic} - x_{jc} & \leq & y_{ij} \\
(C_4)\ \forall i \in \{1,\ldots,n(G)\},\ \forall c \in \{1,\ldots,k\}, & x_{ic} \in \{0,1\} & &
\end{cases}
$$

FIG. 3 – Programme mathématique modélisant la fusion de registres.

modifier les contraintes et la fonction économique du problème en conséquence.

4. Spécification Coq

Cette partie détaille la spécification en Coq de l'algorithme de coloration gourmande, les structures de données utilisées et les théorèmes de correction et d'optimalité de la spécification. Afin de différencier les prédicats et fonctions définis lors du développement de ceux de bibliothèques existantes, les premiers sont en gras et les seconds en italique.

4.1. Définition des graphes

La structure de graphe a été définie en Coq avec la construction `Record`. Deux types de graphes ont été définis : les graphes quelconques et les graphes tels que la liste de leurs sommets est un ordre d'élimination simplicial inverse. La structure générique des graphes est la suivante.

```
Record Graph : Set := mk_Graph{
  vertices : list nat ;
  edges : list (nat×nat) ;
  (P1)  p_is_lex_sorted : is_lex_sorted edges ;
  (P2)  p_is_strict_ord : is_strict_ord edges ;
  (P3)  p_NoDup : NoDup edges ;
  (P4)  p_vertices_edges : vertices = edges_to_vertices edges }.
```

Il a été volontairement choisi de construire tout l'algorithme de coloration à partir uniquement des arêtes du graphe pour s'approcher autant que possible de la structure actuellement utilisée dans CompCert. C'est pourquoi le graphe est modélisé par une liste d'arêtes, une arête étant un couple de sommets. De même, le choix d'utilisation de liste est voulu même s'il conduit à une diminution de complexité d'implantation. En effet, la notion d'ordre d'élimination simplicial est essentielle et se représente idéalement par une liste, laquelle est par définition une permutation de la liste des sommets du graphe.

La liste *edges* des arêtes du graphe possède trois propriétés :
- (P_1) indique que la liste *edges* est triée par ordre lexicographique ;
- (P_2) stipule que la liste *edges* est strictement ordonnée, c'est-à-dire que dans chaque arête, l'identifiant du sommet source est inférieur à celui du sommet destination.
- (P_3) mentionne que la liste *edges* ne contient pas de doublons.

(P_1) et (P_2) servent à améliorer la vitesse des algorithmes. En effet, elles correspondent à des propriétés de tri qui peuvent être exécutées en temps relativement rapide et permettent de diminuer la complexité des algorithmes ou d'y incorporer des conditions d'arrêt. Ces propriétés permettent en outre d'effectuer des parcours parallèles des listes de sommets et d'arêtes du graphe. (P_3) est une propriété de bonne formation du graphe. Toutes les fonctions nécessaires pour transformer la liste initiale ont été implantées et un algorithme de tri rapide a été écrit en Coq à l'aide de la construction `Function` (*cf.* section 4.3).

La liste *vertices* représente le sous-ensemble des sommets du graphe qui possèdent au moins une arête incidente. Le graphe d'interférences étant connexe[5], cet ensemble de sommets est exactement l'ensemble de tous les sommets du graphe. De plus, cette liste est triée par ordre croissant et ne contient pas de doublons, ce qui induit qu'elle est triée par ordre strictement croissant. Ces propriétés découlent de la façon dont a été construite la liste *vertices* à partir de la liste *edges*, c'est-à-dire de la fonction **edges_to_vertices**.

Afin de spécifier ce qu'est un graphe triangulé, il est nécessaire de donner la spécification des ordres d'élimination simpliciaux. Le prédicat (**sv** x v e) signifie que x est un sommet simplicial dans la liste des sommets v par rapport aux arêtes de e. Nous ne donnons pas sa spécification car elle fait elle même appel à d'autres prédicats. La spécification des ordres d'élimination simpliciaux est décomposée en deux définitions **_peo** et **peo** afin d'en alléger l'utilisation.

```
Inductive _peo : list nat → list nat → list (nat×nat) → Prop :=
  peo_cons : ∀ (l vert : list nat) (edg : list (nat×nat)),
                Permutation vert l ⇒
                (∀ (x : nat) (ll rl : list nat), l = ll ++ x : : rl ⇒ sv x (x : : rl) edg) ⇒
                    _peo l vert edg.

Definition peo (l : list nat) (g : graph) : Prop := _peo l (vertices g) (edges g).
```

Le second type de graphes représente les graphes triangulés. Le prédicat (**est_clique** g l t) signifie que les sommets de la liste l forment une clique de taille t dans le graphe g. Ce type de graphe est donc celui où l'ordre inverse des sommets est simplicial. Ainsi, pour tout x l'ensemble des sommets qui précèdent x dans la liste *vertices* (autrement dit ceux qui sont colorés avant lui par l'algorithme de coloration gourmande) forme une clique.

```
Record Chordal_graph : Set := mk_Chordal_graph{
  gph : Graph;
  self_peo : ∀ (x : nat), In x (vertices gph) ⇒
                is_clique gph (x : : get_nghbs gph x) (length (get_nghbs gph x) + 1)}.
```

4.2. Description générale de l'implantation

L'algorithme de coloration gourmande a été écrit en Coq et son optimalité a été prouvée en Coq sur les graphes triangulés. Pour ce faire, il est nécessaire de construire un graphe *my_chordal_gph* afin de créer un enregistrement de type *Chordal_graph* car c'est sur ce type de graphe que l'algorithme de coloration permet d'obtenir une coloration optimale. La figure 4 résume les différentes étapes de la coloration de graphe.

Il faut donc tout d'abord, à partir de la liste des arêtes du graphe, construire *my_graph* de type *Graph*. Ensuite, si *my_graph* est triangulé alors il admet un ordre d'élimination simplicial. Dans ce cas, il est nécessaire de renommer les sommets de *my_graph* de sorte que la liste inverse de l'ordre d'élimination simplicial trouvé corresponde à la liste $\{1,\ldots n\}$. Dès lors, il est possible de construire un graphe *my_chordal_gph* permettant de définir un enregistrement de type *Chordal_graph*, de réaliser

[5] Un graphe connexe est un graphe pour lequel il est possible de relier toute paire de sommets par une liste d'arêtes telle que deux arêtes consécutives sont adjacentes.

la coloration de *my_chordal_gph* (et la prouver optimale), puis de renommer dans le sens inverse les sommets pour prouver que la coloration obtenue (elle-même renommée de la même façon) est une coloration optimale de *my_graph*. Dans le cas où *my_graph* n'admet pas d'ordre d'élimination simplicial, *my_graph* est coloré, mais la coloration obtenue n'est pas optimale.

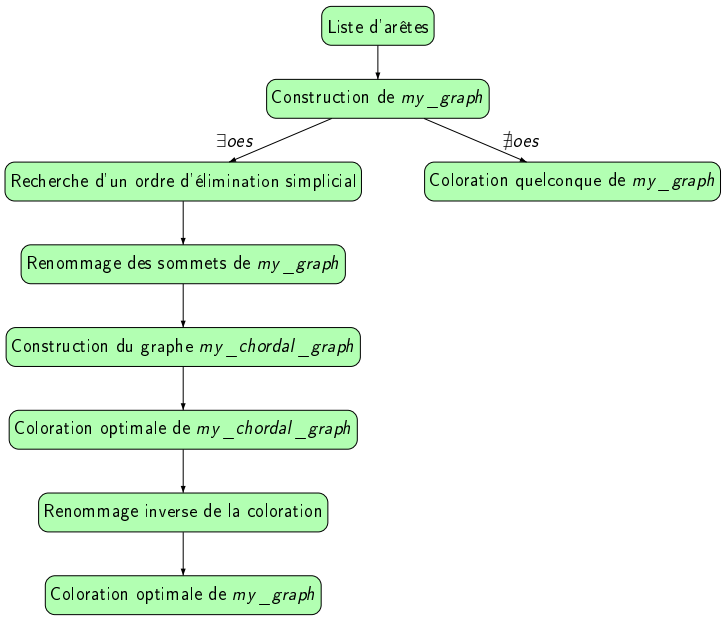


FIG. 4 – Démarche générale de la coloration.

Nous nous sommes attachés à conférer à notre implantation quelques optimisations afin d'obtenir bonne une complexité algorithmique. En particulier, l'utilisation de divers ordres sur les listes permet un gain de complexité appréciable mais allonge considérablement le développement. Les trois principales propriétés d'ordre que nous utilisons sont :

- l'ordre lexicographique sur des listes de couples,
- l'ordre sur les composantes d'un couple (la première composante doit être plus petite que la seconde),
- la croissance des listes d'entiers.

Il a également été nécessaire de définir d'autres ordres connexes à ceux-ci comme les ordres stricts et les ordres inverses. Il existe plusieurs cas de figure pour lesquels l'usage d'ordre procure un gain significatif. Nous présentons ici un cas très simple : l'élimination des doublons d'une liste.

Si l est une liste de longueur lg, alors un algorithme classique d'élimination de doublons se code en $O(lg^2)$. Par contre, si la liste est triée par ordre croissant, il est possible d'effectuer cette opération en $O(lg)$ puisqu'il suffit de vérifier récursivement que les deux éléments de tête de liste sont différents et d'en supprimer un si ce n'est pas le cas. Si la liste n'est pas triée il est donc préférable d'un point de vue de la complexité algorithmique de la trier puis de supprimer les doublons puisque le tri rapide a une complexité en $O(lg\ log(lg))$.

4.3. Recherche d'un ordre d'élimination simplicial

Lorsque le graphe est triangulé, connaître un ordre d'élimination du graphe permet d'obtenir une coloration optimale. Pour le trouver nous utilisons l'algorithme 1 (*cf.* section 3.3). Dans la définition

suivante, la fonction (**sv_search_aux** l l') permet de trouver un sommet simplicial étant donnés les sommets de la liste l et les arêtes de la liste l'. La fonction (**remove** x l) permet de supprimer le sommet x de la liste l, et la fonction (**rm** n l) supprime de la liste l tous les couples dont l'une des composantes est n.

La définition de la fonction **peo_search_aux** n'est pas structurelle puisqu'elle repose sur la décroissance de la longueur de la liste. Aussi nous utilisons la construction `Function`. Il suffit ensuite de prouver que la longueur de la liste décroît bien à chaque itération. La correction de l'algorithme de recherche d'ordre d'élimination simplicial consiste à montrer que s'il existe un ordre d'élimination simplicial pour un graphe alors l'algorithme en trouve un.

```
Function peo_search_aux (l : list nat) (l' : list (nat×nat)) {measure length l} : list nat :=
  match l with nil ⇒ nil
  | _ ⇒ (sv_search_aux l l') : : (peo_search_aux (remove eq_nat_dec (sv_search_aux l l') l)
                                  (rm (sv_search_aux l l') l'))
  end.

Definition peo_search (g : graph) : list nat :=peo_search_aux (vertices g) (edges g).

Lemma peo_peo_search : ∀ (g : graph), (∃ l : list nat, peo l g) ⇒ peo (peo_search g) g.
```

4.4. Algorithme de coloration gourmande

Une fois l'ordre d'élimination simplicial trouvé, il n'est pas encore possible de définir une structure de type *Chordal_graph* puisque l'ordre d'élimination trouvé n'est pas croissant. Il faut donc renommer les sommets de façon à ce que l'ordre d'élimination corresponde à une liste croissante d'entiers. La fonction inverse doit aussi être spécifiée afin de pouvoir renommer les sommets après que la coloration du graphe ait été réalisée. Cette phase est périlleuse mais peu intéressante, c'est pourquoi elle n'est pas détaillée ici. Pour la consulter le lecteur pourra se rapporter à la page web du développement complet.

Une coloration est représentée par une liste de couples (s, c) où c représente la couleur (représentée par un entier) affectée au sommet s. L'algorithme de coloration gourmande construit la coloration au fur et à mesure du parcours de la liste des sommets du graphe. Au moment de la coloration d'un sommet, il est nécessaire de connaître les couleurs affectées aux sommets déjà colorés pour pouvoir choisir la couleur du sommet courant. Il faut donc stocker la coloration dans un accumulateur *col* qui est initialisé par la liste vide. La fonction (**get_available_color** g x col) permet de rechercher dans le graphe g la plus petite couleur n'étant affectée à aucun voisin du sommet x par la coloration partielle *col*.

```
Fixpoint graph_coloring_aux (g : graph) (l : list nat) (col : list (nat×nat)) {struct l} :
            list (nat×nat) :=
  match l with nil ⇒ col
  | x : : l" ⇒ graph_coloring_aux g l" ((x, get_available_color g x col) : : col)
  end.

Definition graph_coloring (g : graph) := graph_coloring_aux g (vertices g) nil.
```

4.5. Propriétés prouvées

Nous avons prouvé en Coq deux familles de propriétés concernant d'une part la correction d'une coloration, et d'autre part l'optimalité de l'algorithme de coloration gourmande dans les graphes triangulés.

Le lemme **is_coloring_graph_coloring** est le lemme de correction de la coloration gourmande.

Il établit que pour tout graphe g, la coloration calculée par la fonction de coloration gourmande **graph_coloring** est bien une coloration valide de g. Une coloration valide est définie par le prédicat **is_coloring**. Soit une coloration *col* d'un graphe g. Soit k tel que les couleurs de *col* sont numérotées de 1 à k. Alors, *col* est valide si et seulement si tout sommet de g possède une et une seule couleur comprise entre 1 et k (lignes (1) à (3)), et si tout couple de sommets formant une arête d'interférence est coloré par deux couleurs distinctes (ligne (4)).

```
Inductive is_coloring : graph → list (nat×nat) → Prop :=
  coloring_cons : ∀ (g : graph) (col : list (nat×nat)),
  (1)      (∀ (x : nat), In x (vertices g) ⇔ ∃ c, In (x,c) col)⇒
  (2)      no_fst_dup col ⇒
  (3)      (∀ (x cx : nat), In (x,cx) col ⇒ 1 ≤ cx) ⇒
  (4)      (∀ (x y cx cy : nat), In (x,y) (edges g)⇒ In (x,cx) col ⇒ In (y,cy) col ⇒ cy ≠ cx) ⇒
      is_coloring g col.

Lemma is_coloring_graph_coloring : ∀ (g : graph), is_coloring g (graph_coloring g).
```

De plus, nous validons *a posteriori* les colorations calculées par le solveur externe que nous avons utilisé. Étant donné un graphe g, la validation *a posteriori* d'une coloration *col* calculée par un solveur externe consiste à vérifier en Coq le lemme **is_coloring** *g col*. Nous vérifions ainsi la même propriété que celle actuellement vérifiée dans CompCert.

Le lemme suivant est utile pour prouver l'optimalité de la coloration gourmande. Soit *my_peo* l'ordre d'élimination simplicial ayant été trouvé, et *my_chordal_gph* le graphe triangulé (de la structure de type Chordal_graph) obtenu à partir de *my_peo* après renommage des sommets. Le lemme **is_coloring_renaming** établit que si *col* est une coloration du graphe triangulé *my_chordal_graph*, alors la coloration obtenue par renommage de la coloration *col* est une coloration du graphe initial *my_graph*. Dans ce lemme, **coloring_renaming** est la fonction qui renomme une coloration afin de rétablir la numérotation des sommets du graphe d'origine.

```
Lemma is_coloring_renaming : ∀ (col : list (nat×nat)),
        is_coloring my_chordal_gph col ⇒
        is_coloring my_graph (coloring_renaming col (rev my_peo)).
```

Le lemme **coloration_optimality** établit l'optimalité de la coloration gourmande. L'optimalité consiste à vérifier que toute coloration valide du graphe utilise au moins autant de couleurs que celle renvoyée par l'algorithme, ou de manière équivalente que la plus grande couleur utilisée par toute coloration est supérieure à la plus grande couleur renvoyée par la coloration de l'algorithme. La fonction **max_color** renvoie le maximum des deuxièmes composantes d'une liste d'entiers (donc ici la couleur maximale de la coloration).

```
Lemma coloring_optimality : ∀ (col : list (nat×nat)),
        is_coloring my_graph col→
        max_color (coloring_renaming (graph_coloring my_chordal_gph) (rev my_peo)).
           ≤ max_color col.
```

La preuve [Wes00] repose sur la propriété suivante : Si $(x_1, \ldots x_{n(G)})$ est un ordre d'élimination simplicial de G, et si l'algorithme de coloration gourmande est appliqué selon l'ordre des sommets $(x_{n(G)}, \ldots x_1)$ alors la coloration est optimale.
Soit G un graphe triangulé et x un ordre d'élimination simplicial de G. Soit i appartenant à $\{1, \ldots n(G)\}$. x étant un ordre d'élimination simplicial, x_i est un sommet simplicial du graphe induit par $\{x_i, \ldots x_n\}$ c'est-à-dire du graphe induit par les sommets déjà colorés et x_i. Ainsi, le graphe induit par x_i et ses voisins déjà colorés est une clique. Soit t_i la taille de cette clique. La couleur affectée à x_i est donc inférieure ou égale à t_i. Cette inégalité étant valide pour tout i, la plus grande couleur

utilisée est inférieure ou égale à la taille de la plus grande clique. Ainsi, nous obtenons l'inégalité $\chi(G) \leq \omega(G)$. De plus, l'inégalité inverse est évidente puisqu'il faut au moins autant de couleurs pour colorer G que pour colorer sa plus grande clique induite, ce qui permet de déduire l'égalité de $\chi(G)$ et $\omega(G)$ et donc l'optimalité de la coloration obtenue.

Notre développement Coq représente environ 10000 lignes de code. Les spécifications et les énoncés des preuves représentent respectivement 4% et 12% du développement. Environ 360 lemmes ont été prouvés. La principale difficulté a été de définir de nombreuses structures de données ainsi que des ordres sur ces structures. Le mécanisme d'extraction automatique de Coq a permis de générer un programme Caml effectuant la coloration gourmande d'un graphe triangulé. Ce programme représente 400 lignes de Caml.

Ce développement Coq a été l'occasion d'utiliser la construction `Function` afin de définir des fonctions récursives non structurelles. Enfin, nous avons été confronté à la lenteur du typeur de Coq. En effet, sur certains lemmes, Coq passait de 30 à 60 minutes à valider la fin de la preuve (la ligne `Qed.`).

Conclusion

Cet article a présenté la vérification formelle en Coq d'un algorithme exact de coloration avec préférences de graphe dédié à l'allocation de registres du compilateur certifié CompCert. L'algorithme est composé de deux phases : 1) la coloration gourmande dont nous avons prouvé la correction dans les graphes quelconques ainsi que l'optimalité dans les graphes triangulés, et 2) une étape de programmation linéaire en nombre entiers qui est résolue par un solveur externe et dont le résultat est validé *a posteriori*.

Afin d'améliorer cette validation *a posteriori*, nous cherchons actuellement à spécifier en Coq la notion de programme linéaire (via une bibliothèque dédiée que nous comptons développer), qui serait toujours résolu par un solveur externe, mais qui serait vérifié par le programme linéaire de Coq. Il s'agit donc de construire la coloration de façon interne à Coq, et de prouver en Coq que toute solution valide du programme linéaire correspond à une coloration valide du graphe d'interférences.

À plus long terme, nous souhaitons vérifier formellement d'autres méthodes de recherche opérationnelle, qui seraient utiles pour améliorer notre allocation de registres et faciliter l'utilisation des méthodes d'optimisation dans les développements de programmes certifiés.

Références

[AG01] Andrew W. Appel and Lal George. Optimal spilling for CISC machines with few registers. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 243–253, 2001.

[AG05] Andrew W. Appel and Lal George. 27,921 actual register-interference graphs generated by standard ML of New Jersey, version 1.09 – `http ://www.cs.princeton.edu/~appel/graphdata/`, 2005.

[And03] Christian Andersson. Register allocation by optimal graph coloring. In *Compiler Construction (CC)*, pages 33–45, 2003.

[BCT94] Preston Briggs, Keith D. Cooper, and Linda Torczon. Improvements to graph coloring register allocation. *Transactions on Programming Languages and Systems (TOPLAS)*, 16(3) :428 – 455, 1994.

[BDL06] Sandrine Blazy, Zaynah Dargaye, and Xavier Leroy. Formal verification of a C compiler front-end. In *FM 2006 : Int. Symp. on Formal Methods*, volume 4085 of *Lecture Notes in Computer Science*, pages 460–475. Springer, 2006.

[Cha82] G J Chaitin. Register allocation and spilling via graph coloring. *Symposium on Compiler Construction*, 17(6) :98 – 105, 1982.

[FG65] D.R. Fulkerson and O.A. Gross. Incidence matrices and interval graphs. *Pac. J. Math.*, 15 :835–855, 1965.

[GA96] Lal George and Andrew W. Appel. Iterated register coalescing. *ACM Trans. Program. Lang. Syst.*, 18(3) :300–324, 1996.

[Gav72] Fănică Gavril. Algorithms for minimum coloring, maximum clique, minimum covering by cliques, and maximum independent set of a chordal graph. *SIAM J. Comput.*, 1 :180–187, 1972.

[GH07] Daniel Grund and Sebastian Hack. A fast cutting-plane algorithm for optimal coalescing. volume 4420 of *Lecture Notes in Computer Science*, pages 111–125. Springer, 2007.

[GW96] David W. Goodwin and Kent D. Wilken. Optimal and near-optimal global register allocations using 0-1 integer programming. *Softw. Pract. Exper.*, 26(8) :929–965, 1996.

[Ilo02] Ilog. Ilog ampl cplex system, version 8.0, user's guide, 2002.

[Ler06] Xavier Leroy. Formal certification of a compiler back-end or : Programming a compiler with a proof assistant. *33rd symposium Principles of Programming Languages*, pages 42–54, 2006.

[NPP07] V. Krishna Nandivada, Fernando Magno Quintão Pereira, and Jens Palsberg. A framework for end-to-end verification and evaluation of register allocators. In *Static Analysis, 14th Int. Symp., SAS 2007, August, 2007, Proc.*, volume 4634 of *Lecture Notes in Computer Science*, pages 153–169. Springer, 2007.

[Oho04] Atsushi Ohori. Register allocation by proof transformation. *Science Computer Programming*, 50(1-3) :161–187, 2004.

[PP05] Fernando Magno Quintão Pereira and Jens Palsberg. Register allocation via coloring of chordal graphs. *Programming Languages and Systems, 3rd Asian Symp., APLAS 2005, Japan, November, 2005, Proc.*, 3780 :315–329, 2005.

[Wes00] Douglas B. West. *Introduction to Graph Theory (2nd Edition)*. Prentice Hall, August 2000.

De la webradio lambda à la λ-webradio

D. Baelde[1] & S. Mimram[2]

1: INRIA & LIX, École Polytechnique
david.baelde@ens-lyon.org
2: Équipe PPS, CNRS / Université Paris 7
Case 7014, 75205 Paris cedex 13
samuel.mimram@ens-lyon.org

Résumé

La génération et la manipulation de flux audio – pour une radio *web* par exemple – est une tâche complexe, difficilement réalisable à l'aide des langages de programmation habituels. Nous présentons dans cet article un langage fonctionnel fortement typé appelé Liquidsoap qui offre des abstractions confortables pour décrire la construction de flux élaborés. Il se démarque par sa souplesse d'utilisation et la richesse des possibilités qu'il offre : de l'utilisation de divers types d'entrées (fichiers audio, micro, requêtes d'utilisateurs) que l'on peut sélectionner dynamiquement (selon la disponibilité ou encore l'horaire) à la gestion des transitions entre morceaux et autres traitements audio. La nécessité d'avoir un langage riche et abordable nous a amenés à introduire une variante du λ-calcul typé, avec étiquettes et arguments optionnels, dont la portée va au delà du domaine du traitement audio.

Avec l'avènement des réseaux à haut débit, il est maintenant possible de diffuser rapidement de grandes quantités de données à travers le monde. Ainsi, de nombreuses radios *web* ont pu voir le jour et diffusent en continu divers contenus sonores par le biais d'Internet. Il n'existait cependant pas de langage simple et expressif permettant de construire les flux audio de ces radios.

Au premier abord, la génération d'un flux continu de données audio peut sembler être une tâche simple à réaliser : il suffit de lire bout à bout des fichiers audio. Cependant, sa mise en pratique se heurte rapidement à certaines difficultés.

- Ces difficultés sont d'abord d'ordre purement technique : les fichiers audio sont stockés sous divers formats (il faut les convertir en un format uniforme), sur divers serveurs (il faut des outils gérant les protocoles utilisés), etc.
- Ensuite, la gestion des listes de lecture, ou *playlists*, s'avère complexe : on veut pouvoir choisir un morceau correspondant à certains critères (le titre, l'artiste, le genre, etc.) dans une base de données, ces critères dépendant de l'horaire (on veut jouer de la musique douce le matin et plus dansante le soir), on veut aussi pouvoir avoir des interventions en direct lors de plages horaires réservés aux animateurs, insérer régulièrement des messages rappelant le nom de la radio (*jingles*), etc.
- Enfin, il faut pouvoir traiter le son provenant des fichiers audio afin de le rendre uniforme et agréable à l'écoute : il faut à la fois appliquer des effets audio sur le son (en particulier normaliser et compresser le son afin d'avoir un volume moyen constant), gérer l'enchaînement entre les morceaux (par exemple, appliquer un fondu enchaîné entre les chansons), éviter les blancs, etc.

Il existait déjà plusieurs logiciels qui permettent de gérer des radios. Les solutions les plus professionelles (Master Control, WinRadio, Open Radio ou encore Rivendell) sont des applications graphiques intégrant la gestion de la grille de diffusion, le contrôle des points de transitions entre fichiers, le décrochage vers une émission en direct et enfin la diffusion. D'autres applications plus spécialisées et légères, ont une interface en mode console et sont plus adaptées à un fonctionnement complètement automatisé sur un serveur. Ces derniers générateurs de flux (par exemple Ices ou

EzStream) offrent des possibilités limitées à la diffusion d'une suite de fichiers sans transitions ou d'un flux en provenance de la carte son. Ils sont cependant très utilisés dans la communauté *open-source*, notamment dans le système Mediabox 404 qui offre une interface web conviviale permettant de gérer une grille de diffusion et le décrochage vers les émissions en direct. Dans tous les cas on remarque que la génération du flux se fait selon un schéma rigide : l'ordre dans lequel les opérations sont effectuées est fixe. Ces outils ne sont par conséquent plus utilisables dès que l'on sort du cadre pour lequel ils ont été conçus.

La conception d'un langage applicatif dédié, fournissant des opérations élémentaires sur des valeurs représentant les flux, offre un cadre de développement riche ainsi qu'un moyen simple et expressif pour l'utilisateur de décrire sa configuration. Nous présentons ici le langage Liquidsoap [5], une implémentation de cette idée en OCaml [7]. Cet outil offre de larges possibilités et est d'ores et déjà utilisé avec succès en production par des webradios [1, 2] ou encore pour expérimenter de nouvelles méthodes de diffusion, par exemple fondées sur le recoupement des habitudes musicales de groupes d'auditeurs [4].

L'une des contraintes majeures qu'un tel langage doit respecter, au delà de celles induites par la manipulation de flux audio, est liée à la nature des utilisateurs potentiels : les personnes susceptibles de vouloir créer des radios sur internet sont loin d'être toutes des programmeuses chevronnées. Nous avons donc conçu le langage de sorte qu'il soit abordable et simple d'utilisation. En particulier, le grand nombre de paramètres dont peuvent dépendre les opérations du langage nous a amené à introduire un calcul avec étiquettes (ce qui permet à l'utilisateur de ne pas avoir à se souvenir de l'ordre des arguments) et arguments optionnels (permettant de spécifier des valeurs par défaut pour les arguments) ainsi qu'un système de types pour ce calcul, qui sont deux contributions originales de cet article. Ce calcul n'est pas spécifique au traitement de l'audio et peut être réutilisé dans d'autres domaines où la simplicité du langage passe avant la nécessité d'une compilation efficace, par opposition avec les calculs développés par Aït-Kaci, Garrigue et Furuse [3, 9].

Nous abordons dans cette article deux aspects fondamentaux de la conception de Liquidsoap. Nous commençons par présenter les abstractions fournies dans le langage en illustrant les possibilités qu'elles offrent et nous discutons de l'adéquation entre ces abstractions et l'implémentation. Dans un second temps, nous formalisons le calcul sous-jacent au langage de programmation ainsi qu'un système de types adapté.

1. Un langage de manipulation de flux audio

Notre présentons ici la méthodologie et les concepts généraux importants qui sous-tendent le langage. Nous nous sommes efforcés d'être synthétiques, le lecteur pourra trouver plus de détails pratiques sur le site dédié au langage [5]. Il est cependant intéressant de donner tout d'abord une rapide idée de l'ampleur du développement qui a été nécessaire pour implémenter ce langage.

Le développement de Liquidsoap a été effectué au sein d'un projet appelé Savonet qui, outre le langage de programmation Liquidsoap lui-même, contient des librairies en OCaml interfaçant des libraires C préexistantes qui permettent de décoder et d'encoder des fichiers audio aux formats Ogg/Vorbis, MP3 et AAC, d'appliquer des effets audio (greffons LADSPA pour les effets audio, samplerate pour changer la fréquence d'échantillonage, etc.), de communiquer avec les cartes son (librairies AO, ALSA et portaudio), avec des programmes externes (JACK), ou d'envoyer le flux audio à un serveur d'émission audio utilisant le protocole SHOUTcast (Icecast par exemple). D'autres programmes de Savonet permettent de créer une base de données des fichiers audio disponibles sur un réseau ou de gérer les requêtes d'utilisateurs via un site web ou un bot IRC – on peut par exemple dire « mets de la techno » sur un forum de messagerie instantanée et une chanson de techno, trouvée dans la base de donnée, sera diffusée sur la radio.

Le projet dans son ensemble comporte plus de 50 000 lignes de code dont 40 000 sont écrites en

OCaml, et 8 000 en C. Le langage Liquidsoap lui-même comporte 25 000 lignes en OCaml. Malgré le lourd travail d'interfaçage de librairies C, le choix d'OCaml semble avoir été fortement positif : l'expressivité et les capacités d'abstraction offertes par le langage nous ont permis de structurer fortement Liquidsoap, de le maintenir et de l'étendre à moindre coût. En particulier, l'utilisation intensive de la programmation objet nous a permis une grande simplicité et extensibilité de la conception des opérations sur les flux.

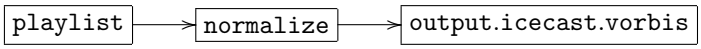
1.1. Génération de flux audio dans Liquidsoap

Un programme Liquidsoap construit des générateurs de flux audio : les *sources*. Une source peut être décrite par un graphe orienté acyclique, dont les sommets sont appelés *opérateurs*. Les sommets initiaux de ce graphe génèrent un flux à partir d'un fichier audio, d'une liste de lecture ou encore d'un microphone. Les sommets internes correspondent à des manipulations opérées sur les flux produits par les sources filles, par exemple la superposition de ces flux, ou le choix entre l'un d'entre eux en fonction de certains paramètres comme l'horaire ou encore l'application d'un effet audio. Les sommets terminaux n'ont typiquement qu'une entrée, et transmettent par exemple leur flux à un serveur de diffusion sur Internet ou à une carte son. Les opérateurs, en plus de dépendre d'autres sources décrites dans le graphe, peuvent dépendre de valeurs données en paramètres (booléens, entiers, flottants, fonctions, etc.).

À partir d'un tel graphe, décrit par un programme Liquidsoap, un flux audio est généré par blocs de données audio. Régulièrement, un bloc est décodé à partir d'une source, des opérateurs procèdent à diverses manipulations sur ce bloc, puis il est encodé dans un format compressé avant d'être transmis à un serveur qui se charge de diffuser le flux aux auditeurs. Par exemple, le programme suivant lit des morceaux dans une liste de lecture appelée `liste` puis applique une normalisation de volume et enfin envoie le flux à un serveur de diffusion :

```
l = playlist("liste")
s = normalize(l)
output.icecast.vorbis(host="www.radio.com", name="ma_radio", s)
```

Le graphe induit par ce script est :

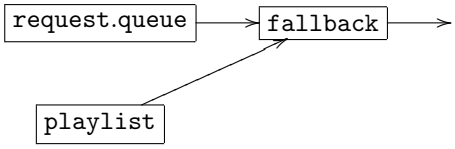


Il nous faut enrichir la notion de flux audio pour pouvoir décrire les configurations usuellement rencontrées. D'une part, certaines sources ne sont pas *disponibles* en permanence, c'est-à-dire qu'elles ne sont pas tout le temps prêtes à générer des données. Par exemple, une source jouant des requêtes d'auditeurs n'est disponible que s'il y a effectivement des requêtes. Lorsque ce n'est pas le cas, il faut pouvoir jouer à la place le flux provenant d'une autre source, typiquement une liste de lecture. Si la source de requêtes devient disponible à nouveau, on ne veut en général pas interrompre le morceau en cours mais attendre qu'il soit terminé avant de jouer la nouvelle requête. Il faut donc d'autre part introduire une notion de *piste* dans le flux qui permette de délimiter les portions faisant partie d'un même morceau.

Liquidsoap permet ainsi de décrire des opérateurs plus complexes comme l'opérateur de choix par défaut `fallback`, qui prend en argument une liste de sources et produit en sortie une piste de la première source disponible, puis à la fin de celle-ci une nouvelle piste de la première source disponible à ce nouvel instant et ainsi de suite. Il permet de réaliser ainsi notre exemple :

```
f = fallback([request.queue(), playlist("liste")])
```

Le graphe sous-jacent à ce programme est :



Le langage permet de construire des sources encore plus élaborées. Nous présentons dans la section suivante l'exemple de l'utilisation des transitions entre pistes, car elle nous semble être une bonne illustration de l'expressivité du langage et motive son caractère fonctionnel.

1.2. Les transitions

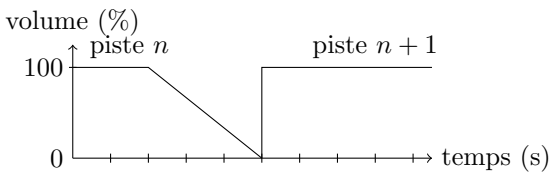
L'opération de fondu enchaîné consiste à faire varier progressivement le volume de 0% à 100% (resp. de 100% à 0%) en début (resp. en fin) de piste. Le fondu enchaîné et croisé consiste de plus à superposer une portion de la fin d'une piste avec le début de la précédente. Cet effet est communément utilisé pour rendre l'écoute plus agréable. De nombreux paramètres sont à prendre en compte : la durée du fondu en début et en fin de piste, la durée de la superposition ou le type de fondu (linéaire ou logarithmique par exemple), etc. De plus, on veut parfois adapter ces paramètres en fonction des volumes sonores, par exemple pour éviter de masquer un début de piste doux en le mixant avec une fin bruyante. Enfin, il est aussi fréquent d'ajouter un *jingle* durant la transition.

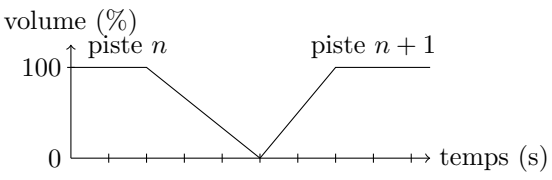
Une solution élégante et générale pour traiter tous ces cas est de décrire une transition par une fonction, qui prend en argument deux sources représentant les pistes à combiner et retourne une source qui est le résultat de la transition. Les opérations usuelles sur les flux sont alors à la disposition de l'utilisateur pour décrire sa transition.

Par exemple, le code suivant permet d'effectuer une transition simple entre deux pistes consécutives d'une source `s` :

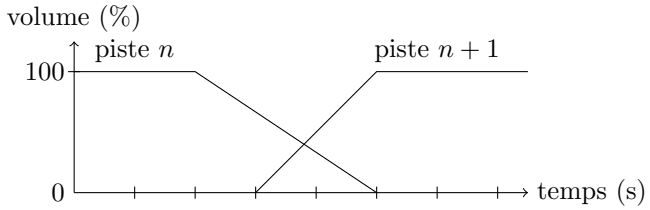
```
s = fade.out(duration=3., s)
s = fade.in(duration=2., s)
fader = fun (a,b) -> add([a,b])
cross(duration=2., fader, s)
```

Dans ce programme, l'opérateur `fade.out` applique une diminution de volume linéaire durant les trois dernières secondes d'une piste, pour arriver au volume nul ; l'opérateur `fade.in` agit similairement mais en début de piste. La seconde définition de `s` masque la première, le `=` de la syntaxe concrète étant à lire comme un `let` dont le `in` est implicite. La fonction `fader` est ensuite définie pour décrire comment va se faire la transition entre les pistes. Son premier paramètre sera un tronçon de flux à croiser en fin de piste, le second paramètre correspondra au flux commençant à la piste suivante. Ici, on se contente de superposer les deux pistes grâce à l'opérateur `add` mais on aurait aussi pu ajouter un *jingle* durant la transition en utilisant la fonction `fader = fun (a,b) -> add([a,once(jingles),b])`. Enfin, l'opérateur `cross` va jouer le flux en appliquant la transition à chaque changement de piste : à deux secondes de la fin d'une piste, la fonction de transition est utilisée pour calculer l'enchaînement des pistes. Graphiquement, après la première et la deuxième ligne du script, le volume de deux pistes consécutives sera respectivement :





Enfin, à la fin du script, après l'utilisation de l'opérateur `cross`, l'enchaînement entre pistes sera schématiquement :



L'opérateur `cross` doit fournir en même temps à sa fonction `fader` des données provenant de la même source `s` mais correspondant à des instants différents dans le flux. Il doit donc récupérer en avance et stocker les données à superposer. Si un autre opérateur `t` accédait à la source `s`, il faudrait stocker ses anciennes données pour pouvoir fournir des réponses cohérentes à `t`. Sans hypothèse simplificatrice, ceci est irréalisable : la quantité de données à stocker croît sans cesse. Dans Liquidsoap, on choisit d'exclure complètement cette possibilité : aucune des sources en amont d'un opérateur `cross` ne doit être accédée par un autre opérateur. Malheureusement, c'est encore à l'utilisateur de vérifier cette condition, en attendant une extension adéquate du système de types (cf. Section 3.2).

Pour finir, notons que cette représentation des transitions motive pleinement la conception de Liquidsoap comme un langage fonctionnel : contrairement à de nombreux langages spécialisés (*Domain Specific Languages*), la notion de fonction est essentielle dans Liquidsoap. En effet, on ne peut pas éliminer les fonctions par une première passe d'évaluation partielle, car leur exécution est étroitement liée au processus infini de génération de flux.

1.3. Granularité, efficacité et partage

Nous avons décrit un système expressif pour construire un flux, en composant des opérations élémentaires prédéfinies. Nous discutons ici de l'écart entre la notion abstraite de flux que l'utilisateur manipule, et son implémentation.

Précisons le cadre dans lequel se situe Liquidsoap, ses objectifs, sa *granularité*, par comparaison avec les *langages synchrones* [6] dans lesquels l'évaluation se fait régulièrement, et de façon théoriquement instantanée, à chaque tic d'une horloge globale. Il existe de tels langages dédiés à la synthèse audio, ChucK [10] ou StreamIt [11] par exemple. Ces langages offrent une granularité très fine : ils permettent à l'utilisateur un contrôle précis du flux, au niveau de l'échantillon. La synthèse sonore s'effectue échantillon par échantillon, en exécutant périodiquement le programme saisi par l'utilisateur. Une telle richesse n'est ni nécessaire ni souhaitable dans notre cas. Le créateur de radio *web* n'a pas besoin d'implémenter un nouvel algorithme de synthèse sonore ou un filtre innovant ; il n'en a en général d'ailleurs pas les capacités techniques et préfère en tout cas s'abstraire de ces détails de bas niveau. Liquidsoap offre donc une granularité plus grossière : le langage ne donne pas accès aux flux mais traite ceux-ci comme des objets complètement abstraits. L'exécution du programme crée essentiellement une source en assemblant des boîtes noires dont le comportement est programmé en OCaml et non pas dans le langage Liquidsoap. La synthèse du flux audio n'a lieu que dans un second temps, une fois la source créée, et n'implique (presque) plus l'évaluation du programme Liquidsoap saisi par l'utilisateur.

La simplicité d'utilisation n'est cependant pas l'unique motivation de la conception de Liquidsoap qui est aussi guidée par une recherche d'efficacité. En effet, la majeure partie des calculs est ainsi effectuée en OCaml qui est un langage efficacement compilé, mais surtout le niveau d'abstraction permet un traitement du flux optimisé en interne. Au lieu de calculer le flux échantillon après échantillon, les opérations définies en OCaml vont pouvoir travailler directement avec des blocs d'échantillons, c'est-à-dire un tableau d'échantillons consécutifs de taille fixe. Ceci évite de nombreuses copies de données et est crucial pour les performances : par exemple, en passant d'une taille de bloc

de 1 à 10 échantillons on constate une accélération d'un facteur 5, puis encore de 2 en passant de 10 à 100.

Cependant, cette optimisation des copies a une conséquence sur le flux calculé. Il se peut qu'à un instant donné un opérateur demande à une de ses sources filles de remplir un bloc d'échantillons à partir d'une position quelconque qui n'est pas nécessairement au début du bloc. Par exemple, un opérateur de choix par défaut `fallback` va, si l'une de ses sources se tarit, finir de remplir un bloc avec une autre source. On rappelle par ailleurs que le graphe décrivant un flux n'étant pas nécessairement un arbre, une même source S peut être *partagée* par deux opérateurs A et B. En général, il est impossible pour S de savoir au début d'un instant lequel de ces opérateurs va effectivement lui réclamer des données, car cela dépend notamment des données générées par les autres sources dans le même instant. La source S doit donc se préparer à tous les scénarios. À un instant donné, A peut lui demander de remplir un bloc, à partir du milieu. Puis, dans le même instant, B peut demander à S des données, cette fois à partir du début d'un bloc. Or, S doit produire des résultats cohérents : même un décalage d'un demi-bloc entre les flux envoyés à A et B est intolérable, par exemple dans le cas où ces flux sont ensuite combinés. La source S se voit donc contrainte de générer un bloc complet, et d'en copier la seconde moitié pour A. Si B réclame effectivement des données, on pourra les copier à partir de notre bloc complet. Sinon, on aura généré un début de bloc inutile, ce qui crée une légère différence entre le flux qui aurait été produit par un traitement échantillon par échantillon.

Afin de minimiser cet écart entre l'implémentation et l'abstraction que l'utilisateur manipule, on doit chercher à limiter les effets du partage, c'est-à-dire détecter le plus finement possible les points de partage potentiel. Par ailleurs, cela entraine une limitation des copies de blocs, donc encore une optimisation – cette fois sans effet sur le résultat. La détection du partage devient plus délicate en présence de transitions, mais nous ne la détaillerons pas ici.

2. Un langage de programmation étiqueté

L'utilisation d'un langage de programmation *fonctionnel* pour décrire les graphes de configuration apparait naturellement, les flux étant construits à l'aide d'opérateurs ayant plusieurs entrées et au plus une sortie. De plus, cela permet de décrire de façon élégante des constructions complexes comme les transitions (voir Section 1.2). Enfin, l'*application partielle* s'est aussi révélée être pratique pour factoriser le code, y compris au sein de scripts simples.

Nous avons voulu un langage *fortement* et *statiquement typé*. La garantie d'absence d'erreur à l'exécution que cela apporte nous semble d'autant plus importante ici que les scripts sont amenés à fonctionner pendant de très longues durées sans maintenance : il serait malheureux qu'une coquille dans le code d'une transition spécifique au troisième dimanche du mois provoque l'arrêt de toute la diffusion. D'autre part, le typage fournit un support utile à la documentation, comme on peut le voir pour l'opérateur `cross`, déjà rencontré à la Section 1.2 :

```
$ liquidsoap -h cross
Generic cross operator, allowing the composition of the N last seconds of a track
with the beginning of the next track.
Type: (?duration:float, ((source, source)->source), source)->source
Parameters:
* duration    :: float (default 5.)                Duration of the composition (s).
* (unlabeled) :: (source, source)->source                        Transition function.
* (unlabeled) :: source
```

Les opérateurs prédéfinis de Liquidsoap dépendent souvent de nombreux paramètres. Par exemple, la sortie la plus couramment utilisée, qui envoie le flux encodé au format Ogg/Vorbis à un serveur de diffusion, dépend de vingt paramètres. Pour que le langage reste attractif et utilisable simplement, nous

avons donc été conduits à utiliser des *arguments étiquetés*, ce qui évite d'avoir à se souvenir de l'ordre des arguments. De plus, de nombreux paramètres ayant des valeurs par défaut raisonnables, nous avons ajouté la possibilité d'avoir des *arguments optionnels*, qui permettent d'utiliser les valeurs par défaut quand aucune valeur n'a été spécifiée. Ainsi, dans l'exemple précédent, le paramètre spécifiant la durée de la superposition est étiqueté `duration` et est optionnel, avec comme valeur par défaut 5 secondes.

2.1. Un langage orienté vers la simplicité

Plusieurs articles établissent les fondations d'un λ-calcul avec étiquettes [3] et arguments implicites [9], et ont mené à l'implémentation de ces traits dans le langage OCaml. Avant d'introduire le calcul avec étiquettes et arguments implicites utilisé dans Liquidsoap, mentionnons brièvement pourquoi nous avons trouvé utile de nous démarquer de [9], en soulignant les difficultés que posent ce genre de calculs. Les auteurs de cet article s'attachent à la compilation efficace du langage, ce qui entraîne certaines restrictions peu naturelles pour l'utilisateur non averti. Par exemple en OCaml, les arguments étiquetés ne commutent pas toujours :

```
# let app f = f ~a:1 ~b:2 ;;
val app : (a:int -> b:int -> 'a) -> 'a = <fun>
# app (fun ~b ~a -> a+b) ;;
This function should have type a:int -> b:int -> 'a
but its first argument is labeled ~b
```

Dans la même optique, les auteurs souhaitent implémenter les arguments optionnels comme un « sucre syntaxique typé » : après une première phase de typage, le code appliquant les valeurs par défaut est ajouté implicitement lorsque tous les paramètres obligatoires sont déja appliqués. Ceci limite les abstractions sur une fonction prenant un argument implicite. Considérons par exemple la fonction `fun f x -> f x`. Avec le type `('a -> 'b) -> 'a -> 'b`, inféré par OCaml, cette fonction sera compilée sans prendre en compte la possibilité d'arguments optionnels à substituer implicitement dans `f`, et le système de types interdira de passer une fonction avec un argument optionnel. En revanche, si on force le type de `f` à être par exemple `?a:int -> int -> int`, la fonction sera compilée pour implicitement substituer l'argument étiqueté `a`, mais n'acceptera plus de fonction sans argument implicite. Dans [9], les auteurs obtiennent un système de type et un algorithme qui infère des types principaux (cf. Section 2.4), mais ils doivent pour cela interdire toute abstraction sur une fonction avec argument implicite.

En présence d'arguments optionnels, on notera le besoin de distinguer entre les applications successives d'une fonction à des arguments et la *multi-application* d'une fonction à des arguments c'est-à-dire l'application de la fonction à plusieurs arguments *simultanément*. Ceci est nécessaire pour contrôler le moment où les arguments optionnels sont implicitement appliqués. L'exemple suivant le montre avec OCaml :

```
# let f ?(a=false) () = a ;;
val f : ?a:bool -> unit -> bool = <fun>
# f () ~a:true ;;
- : bool = true
# (f ()) ~a:true ;;
This expression is not a function, it cannot be applied
```

Cependant, la multi-application d'OCaml ne peut être vide (0-aire) : on ne peut pas appliquer une fonction de type `?l:t -> t'` à « rien » pour obtenir un terme de type `t'`, calculé en substituant l'argument optionnel par sa valeur par défaut dans le corps de la fonction. Dans ce cas l'argument optionnel est *ineffaçable*, c'est-à-dire que la valeur par défaut ne peut jamais être utilisée.

Dans notre approche, l'efficacité passe après la simplicité d'utilisation, car les réductions dans le langage prennent généralement un temps négligeable devant les traitements audio. Cela nous a amenés à introduire un calcul légèrement différent de [9]. Nous nous proposons de plus de considérer une notion de multi-abstraction, une approche qu'il paraît naturel d'explorer, par symétrie avec la multi-application. Le système obtenu n'est peut-être pas compilable efficacement, mais sa sémantique est intuitive, et son interprétation simple.

Nous allons donc formaliser un calcul et un système de types originaux. On notera que le cœur du langage n'est pas spécifique au traitement audio et pourrait être réutilisé dans d'autres domaines.

2.2. Termes du langage

On se donne un ensemble L d'*étiquettes* et un ensemble V de *valeurs de base*. En pratique dans Liquidsoap, l'unité, les booléens, les entiers, les flottants et les chaînes de caractères font partie des valeurs de base, mais aussi les sources et les requêtes. On note X l'ensemble des *variables*. Les termes sont définis inductivement par la grammaire suivante :

$$\begin{array}{rcll} M & ::= & v & (1)\\ & | & x & (2)\\ & | & \texttt{let } x = M \texttt{ in } M & (3)\\ & | & \lambda\{\ldots, l_i : x_i, \ldots, l_j : x_j = M, \ldots\}.M & (4)\\ & | & M\{l_1 = M, \ldots, l_n = M\} & (5) \end{array}$$

La multi-application (5) est l'application simultanée d'une fonction à plusieurs arguments étiquetés par des $l_i \in L$. La multi-abstraction (4) abstrait simultanément sur plusieurs arguments en les étiquetant[1] par $l_i \in L$ et en précisant pour certains arguments une *valeur par défaut*. La notation compacte utilisée dans (4) dénote une liste d'arguments quelconque, sans contrainte sur la position relative des arguments avec et sans valeur par défaut. Toujours dans la multi-abstraction, on supposera les x_i deux-à-deux distincts[2]. Les arguments pour lesquels une valeur par défaut est donnée sont appelés *optionnels*, dans le cas contraire ils sont *obligatoires*. On dira enfin qu'un groupe de *liaisons* $\Gamma = \{\ldots, l_i : x_i, \ldots, l_j : x_j = M_j, \ldots\}$ est *effaçable* si toutes les liaisons qu'il contient sont optionnelles (en particulier, l'ensemble vide est effaçable). Dans la multi-abstraction $\lambda\{\Gamma\}.M$, les variables x_i sont liées dans M, mais pas dans les valeurs par défaut M_j. Ceci permet de définir l'ensemble $\mathcal{FV}(M)$ des variables libres d'un terme M ; lorsque cet ensemble est vide, le terme M est dit *clos*.

Nos termes sont considérés modulo la relation d'α-équivalence habituelle. On considérera aussi que deux arguments d'étiquettes distinctes peuvent permuter au sein d'une même multi-abstraction[3]. Formellement, cela revient à considérer les termes modulo la relation $\equiv$ définie comme la plus petite congruence satisfaisant :

$$\left.\begin{array}{rcl} \lambda\{\Gamma, l : x, l' : x', \Delta\}.M & \equiv & \lambda\{\Gamma, l' : x', l : x, \Delta\}.M\\ \lambda\{\Gamma, l : x = N, l' : x', \Delta\}.M & \equiv & \lambda\{\Gamma, l' : x', l : x = N, \Delta\}.M\\ \lambda\{\Gamma, l : x = N, l' : x' = N', \Delta\}.M & \equiv & \lambda\{\Gamma, l' : x' = N', l : x = N, \Delta\}.M \end{array}\right\} \text{ si } l \neq l' \qquad (6)$$

où Γ et Δ sont des listes d'arguments étiquetés, optionnels ou pas. On pourra vérifier par ailleurs que l'ajout de la relation similaire de permutation dans les multi-applications est compatible avec la réduction du calcul.

[1] On représente facilement les arguments non étiquetés, possiblement optionnels, en les étiquetant par une étiquette particulière.

[2] Mais deux étiquettes peuvent être égales, comme on le voit dans l'Exemple 2.

[3] L'implémentation dans Liquidsoap est légèrement différente, car elle permet de définir une valeur par défaut en fonction des valeurs des arguments précédents. Les notions de liaison et de substitution tiennent alors compte de l'ordre des arguments dans une multi-abstraction, mais la réduction se fait toujours modulo permutation. Cela complique l'écriture du système mais ne change essentiellement rien.

L'application d'une fonction $\lambda\{\ldots, l : x, \ldots\}.M$ à un terme N sur l'étiquette l va provoquer une substitution de la variable x par N dans le corps M, et la suppression de la liaison $l : x$ dans la multi-abstraction. Si à l'issue d'une multi-application aucun argument obligatoire ne subsiste, les valeurs par défaut seront automatiquement substituées aux argument optionnels et la multi-abstraction disparaîtra. Afin de formaliser ceci, on introduit la relation de substitution suivante :

$$\begin{aligned}
x[M/x] &= M \\
y[M/x] &= y, \text{ si } x \neq y \\
(\texttt{let } y = N \texttt{ in } P)[M/x] &= \texttt{let } y = N[M/x] \texttt{ in } P[M/x] \\
(\lambda\{\ldots, l_i : y_i, \ldots, l_j : y_j = N_j, \ldots\}.P)[M/x] &= \lambda\{\ldots, l_i : y_i, \ldots, l_j : y_j = N_j[M/x], \ldots\}.(P[M/x]) \\
(N\{\ldots, l_i = P_i, \ldots\})[M/x] &= N[M/x]\{\ldots, l_i = P_i[M/x], \ldots\}
\end{aligned}$$

Pour le cas du `let` on suppose y distinct de x et non libre dans M. De même pour la multi-abstraction, on suppose que les variables y_i sont distinctes de x et non libres dans M. La sémantique opérationnelle de notre calcul est alors définie par les règles de réduction suivantes :

$$\texttt{let } x = M \texttt{ in } N \rightsquigarrow N[M/x] \tag{7}$$

$$(\lambda\{\overrightarrow{l_i : x_i, l_j : x_j = M_j}, \Gamma\}.M)\{\overrightarrow{l_i = N_i}\} \rightsquigarrow \lambda\{\Gamma\}.(M[\overrightarrow{N_i/x_i}]), \text{ si } \Gamma \text{ est ineffaçable} \tag{8}$$

$$(\lambda\{\overrightarrow{l_i : x_i, l_j : x_j = P_j}, \overrightarrow{l'_k : y_k = M_k}\}.M)\{\overrightarrow{l_i = N_i}\} \rightsquigarrow M[\overrightarrow{N_i/x_i}, \overrightarrow{M_k/y_k}] \tag{9}$$

La notation vecteur ci-dessus désigne une liste de liaisons ou de substitutions. En particulier, la notation $\overrightarrow{l_i : x_i, l_j : x_j = M_j}$ représente une liste de liaisons dont certaines ont une valeur par défaut, sans contrainte particulière sur la position de celles-ci dans la liste. De plus, $M[\overrightarrow{N/x}]$ désigne une séquence de substitutions. Dans la règle (8), on suppose que les variables x_i, ainsi que les variables liées par Γ, ne sont pas libres dans les N_i ; de même dans la règle (9), on supposera que les variables x_i et y_k ne sont pas libres dans les N_i et M_k. Les règles (8) et (9) permettent de réduire l'application d'une abstraction à un ensemble de paramètres étiquetés $\{\overrightarrow{l_i = N_i}\}$, ces arguments correspondant à des paramètres optionnels ou obligatoires de l'abstraction. Dans les deux cas, les règles de réduction provoquent la substitution, dans le corps de l'abstraction, des variables x_i associées aux arguments par la valeur du paramètre N_i, ignorant les valeurs par défaut des x_i optionnels – M_i dans (8) et P_i dans (9). Si des valeurs n'ont pas été assignées à tous les arguments obligatoires, c'est la règle (8) qui s'applique, laissant l'abstraction en attente d'une autre application. Sinon, la règle (9) supprime l'abstraction et substitue les variables associées aux arguments optionnels restants $l'_k : y_k = M_k$ par leur valeur par défaut.

Exemple 1. Le terme $\lambda\{l_1 : x, l_2 : y = 12, l_3 : z = 13\}.M$, appliqué à $\{l_3 = 3\}$, se réduit en $\lambda\{l_1 : x, l_2 : y = 12\}.M[3/z]$. On notera que lors de cette réduction la valeur par défaut de z est ignorée. Si on applique le résultat à $\{l_1 = 1\}$, on obtient alors $M[1/x, 12/y, 3/z]$: la variable y a été implicitement substituée par sa valeur par défaut.

Exemple 2. On retrouve dans ce calcul un phénomène similaire aux arguments ineffaçables d'OCaml. Dans le terme $\lambda\{l : x = 3, l : y\}.M$, il est impossible de permuter les deux arguments. Ainsi, la première application sur l'étiquette l désignera l'argument optionnel x : on est contraint de donner une valeur à x avant de pouvoir en donner une à y et permettre la réduction (9). On préférera donc utiliser deux étiquettes distinctes ou mettre l'argument optionnel en seconde position : $\lambda\{l : y, l : x = 3\}.M$. Dans ce dernier cas, l'application d'un unique paramètre sur l désignera l'argument y et permettra la réduction (9), la multi-application de deux paramètres sur l étant toujours possible si l'on veut aussi spécifier une valeur pour x.

Proposition 1. *Le calcul est confluent.*

Dans la pratique, l'implémentation de Liquidsoap utilise une stratégie de réduction en appel par valeur, ce qui a son importance, car certaines fonctions prédéfinies ont des effets de bord.

2.3. Types

Nous décrivons maintenant un système de types pour nos termes. On se donne un ensemble A de *variables de type*, un ensemble B de *types de base* (dans Liquidsoap, il y a par exemple un type de base pour les sources), et une fonction $\mathcal{T} : V \to B$ qui donne le type des valeurs élémentaires. Le type flèche du λ-calcul devient ici une multi-flèche étiquetée :

$$\begin{array}{rcl} t & ::= & \iota \\ & | & \alpha \\ & | & \{\ldots, l_i : t_i, \ldots, ?l_j : t_j, \ldots\} \to t \end{array}$$

Ci-dessus, on suppose $\iota \in B$ et $\alpha \in A$. Les étiquettes annotées $?l$ dans la multi-flèche dénotent les arguments optionnels. De même qu'avec la multi-abstraction, on considérera les types modulo une permutation dans la multi-flèche définie de façon similaire à (6). La substitution $t[t'/\alpha]$ d'une variable de type $\alpha \in A$ par un type t' dans un type t est définie de façon habituelle.

On introduit du polymorphisme prénexe à la Damas-Milner [8], en se donnant la possibilité de quantifier universellement sur une variable de type dans les *schémas de type* :

$$\sigma \quad ::= \quad t \quad | \quad \forall \alpha.\, \sigma$$

Les règles de réduction introduisent la possibilité pour une fonction $\lambda\{\Gamma, \Delta\}.M$, dont les arguments sont Γ, Δ, de se comporter localement comme une fonction dont les arguments sont Γ et qui renvoie une fonction dont les arguments sont Δ. Ceci est exprimé naturellement ici par la relation de sous-typage suivante :

$$\frac{\Delta \text{ ineffaçable}}{\{\Gamma, \Delta\} \to t \leq \{\Gamma\} \to \{\Delta\} \to t} \qquad \frac{\Delta \text{ effaçable}}{\{\Gamma, \Delta\} \to t \leq \{\Gamma\} \to t}$$

$$\frac{}{t \leq t} \qquad \frac{t \leq t' \qquad t' \leq t''}{t \leq t''}$$

$$\frac{t \leq t' \qquad t'_1 \leq t_1 \qquad \cdots \qquad t'_n \leq t_n}{\{\ldots, l_i : t_i, \ldots, ?l_j : t_j, \ldots\} \to t \leq \{\ldots, l_i : t'_i, \ldots, ?l_j : t'_j, \ldots\} \to t'}$$

Remarque 1. Il peut sembler naturel d'ajouter l'inéquation

$$\{\Gamma, ?l : t, \Delta\} \to t' \quad \leq \quad \{\Gamma, l : t, \Delta\} \to t' \tag{10}$$

à la définition de la relation de sous-typage, mais celle-ci est incompatible avec le calcul. En effet, la fonction $f := \lambda\{l : x = 42\}.x$, qui a le type $\{?l : int\} \to int$, ne se comporte pas comme un terme de type $\{l : int\} \to int$. Par exemple, elle ne peut être appliquée successivement à $\{\}$ puis à $\{l = 12\}$: dès la première application, elle se réduit en 42. En revanche, la fonction $g := \lambda\{l : x\}.x$, qui a le type $\{l : int\} \to int$, se réduit en elle-même quand on l'applique à $\{\}$, puis se réduit en 12 quand on l'applique à $\{l = 12\}$.

Les règles de typage de notre calcul sont :

$$\frac{}{\Gamma \vdash v : \mathcal{T}(v)}(\text{Val}) \qquad \frac{}{\Gamma, x : \forall\alpha_1.\ \ldots \forall\alpha_n.\ t \vdash x : t[t_1/\alpha_1, \ldots, t_n/\alpha_n]}(\text{Ax})$$

$$\frac{\Gamma \vdash M : t' \qquad \Gamma, x : \forall\alpha_1.\ \ldots \forall\alpha_n.\ t' \vdash N : t \qquad \{\alpha_1, \ldots, \alpha_n\} = \mathcal{FV}(t') \setminus \mathcal{FV}(\Gamma)}{\Gamma \vdash \texttt{let}\ x = M\ \texttt{in}\ N : t}(\text{Let})$$

$$\frac{\Gamma \vdash M : \{\ldots, l_i : t_i \ldots, ?l_j : t_j, \ldots\} \to t \qquad \Gamma \vdash N_1 : t_1 \qquad \cdots \qquad \Gamma \vdash N_n : t_n}{\Gamma \vdash M\{\ldots, l_i = N_i, \ldots, l_j = N_j, \ldots\} : t}(\text{App})$$

$$\frac{\Gamma, \ldots, x_i : t_i, \ldots, x_j : t_j, \ldots \vdash M : t \qquad \cdots \qquad \Gamma \vdash M_j : t_j \qquad \cdots}{\Gamma \vdash \lambda\{\ldots, l_i : x_i, \ldots, l_j : x_j = M_j, \ldots\}.M : \{\ldots, l_i : t_i, \ldots, ?l_j : t_j, \ldots\} \to t}(\text{Abs})$$

$$\frac{\Gamma \vdash M : t' \qquad t' \leq t}{\Gamma \vdash M : t}(\text{Sub})$$

De même que dans [8], on limite l'introduction de schémas de type à la règle (Let) ; les schémas sont éliminés par instantiation complète dans la règle (Ax). La règle (App) type la multi-application d'une fonction à des arguments qui peuvent aussi bien se substituer à des paramètres optionnels qu'obligatoires. Cette règle sera le plus souvent utilisée en conjonction avec (Sub) pour autoriser les applications partielles et l'effacement des arguments optionnels. La règle (Abs) vérifie d'une part que le corps de la fonction a le type promis si les arguments ont bien le type demandé, d'autre part que les valeurs par défaut des arguments optionnels ont le bon type.

Exemple 3. La fonction $f := \lambda\{a : x, b : y, c : z = 42\}.x$ admet le type $\{a : int, b : int, ?c : int\} \to int$, qui est un sous-type de $\{a : int\} \to \{b : int, ?c : int\} \to int$. On en déduit que la fonction $f' := f\{a = 16\}$ admet le type $\{b : int, ?c : int\} \to int$. Ce dernier est un sous-type de $\{b : int\} \to int$, et par suite $f'\{b = 69\}$ est de type int. De fait, ce terme se réduit en 16.

Proposition 2 (Réduction du sujet)**.** *Si* $\Gamma \vdash M : t$ *et* $M \rightsquigarrow M'$*, alors* $\Gamma \vdash M' : t$*.*

Lemme 1. *Les dérivations de typage vérifient plusieurs propriétés usuelles, exprimées par l'admissibilité des règles suivantes :*

$$\frac{\Gamma \vdash M : t}{\Gamma, x : \sigma \vdash M : t}\textit{(Weak)} \qquad \frac{\Gamma, x : t' \vdash M : t \qquad \Gamma \vdash N : t'}{\Gamma \vdash M[N/x] : t}\textit{(Subst)}$$

$$\frac{\Gamma, x_1 : t_1, \ldots, x_n : t_n \vdash M : t \qquad t \leq t' \qquad t'_1 \leq t_1 \qquad \cdots \qquad t'_n \leq t_n}{\Gamma; x_1 : t'_1, \ldots, x_n : t'_n \vdash M : t'}\textit{(Subst-Sub)}$$

Proposition 3 (Terminaison)**.** *Si* M *est un terme tel que* $\vdash M : t$ *est dérivable alors toute suite de réductions*

$$M \rightsquigarrow M_1 \rightsquigarrow M_2 \rightsquigarrow \ldots$$

est nécessairement finie.

En utilisant la Proposition 1, on en déduit que tout terme typé admet une *forme normale*.

2.4. Inférence de type

L'algorithme d'inférence utilisé dans Liquidsoap est l'algorithme W de Damas et Milner [8] naïvement étendu à notre calcul pour prendre en compte les multi-abstractions et les multi-applications, ainsi que les arguments étiquetés et optionnels.

Notre algorithme étend strictement celui de Damas et Milner dans le sens suivant. Étant donnée une étiquette $l \in L$, tout terme M de ML peut être vu comme un terme $[\![M]\!]$ dans notre langage par l'interprétation définie par

$$[\![v]\!] = v, \qquad [\![x]\!] = x, \qquad [\![\lambda x.M]\!] = \lambda\{l : x\}.[\![M]\!], \qquad [\![MN]\!] = [\![M]\!]\{l = [\![N]\!]\}$$

et comme le morphisme attendu sur les constructions `let`. De même, on peut voir tout type t de ML comme un type $[\![t]\!]$ de notre langage et ces deux interprétations sont injectives et compatibles avec la réduction. On peut alors montrer que si t est le type d'un terme M de ML, inféré par l'algorithme W, alors $[\![t]\!]$ est le type inféré par notre algorithme pour $[\![M]\!]$.

Une propriété majeure de l'algorithme W est qu'il infère un (schéma de) type canonique, appelé *type principal*, pour un terme. Dans notre système, un terme M admet un type principal t si tout autre type t' de M peut se déduire de t par sur-typage ou par une sorte particulière de substitution appelée *instantiation générique*. Malheureusement, un terme n'admet pas nécessairement de type principal. Par exemple, la fonction $f := \lambda\{l : g\}.(g\{l' = 42\})$ admet à la fois

$$t := \forall\alpha.\ l : \{\{l' : int\} \to \alpha\} \to \alpha \qquad \text{et} \qquad t' := \forall\alpha.\ l : \{\{?l' : int\} \to \alpha\} \to \alpha \tag{11}$$

comme types, et aucun de ces deux types n'est sous-type ou instance de l'autre. Ce défaut est essentiellement dû au fait que la règle de sous-typage évoquée dans la Remarque 1 est invalide :

$$\{\Gamma, ?l : t, \Delta\} \to t' \quad \leq \quad \{\Gamma, l : t, \Delta\} \to t' \tag{10}$$

On peut cependant *plonger* les termes de type $\{\Gamma, ?l : t, \Delta\} \to t'$ dans les termes de type $\{\Gamma, l : t, \Delta\} \to t'$. Par exemple, à partir d'un terme f de type $\{?l : int\} \to int$, on peut construire un terme de type $\{l : int\} \to int$ en utilisant une sorte d'η-expansion : $\lambda\{l : x\}.f\{l = x\}$. Dans ce sens, le type t de f donné en (11) est donc plus général que le type t' donné en (11), car on peut transformer les arguments pour le second en arguments pour le premier. Le type t est précisément celui qui sera inféré pour la fonction f par notre algorithme. Ainsi, les types inférés sont canoniques mais dans un sens plus faible que la principalité.

2.5. Une variante du calcul

Nous évoquons ici une variante du calcul qui permet de s'accommoder de la règle de sous-typage (10), rendant ainsi possible l'existence d'un type principal pour tous les termes. Nous ne l'avons pas choisie lors de l'implémentation de Liquidsoap, car si elle pourrait mieux se prêter à une étude théorique, elle est moins naturelle à utiliser dans un langage de programmation et le typage qu'elle nécessite semble plus complexe à définir et à mettre en œuvre.

Au lieu d'avoir une application et deux règles de réduction, l'une pour l'application partielle et l'autre pour l'application totale, décomposons l'application en deux constructions[4] : l'*application partielle* $M\{l_1 = M_1, \ldots, l_n = M_n\}$ et la *destruction des multi-abstractions* dont les liaisons sont optionnelles $M\bullet$, avec les règles de réduction suivantes.

$$\begin{aligned}(\lambda\{\overrightarrow{l_i : x_i, l_j : x_j = M_j}, \Gamma\}.M)\{\overrightarrow{l_i = N_i}\} &\rightsquigarrow \lambda\{\Gamma\}.(M[\overrightarrow{N_i/x_i}])\\ (\lambda\{\overrightarrow{l_i : x_i = M_i}\}.M)\bullet &\rightsquigarrow M[\overrightarrow{M_i/x_i}]\end{aligned}$$

Dans la première règle, on n'impose plus que le groupe de liaison Γ ne soit pas effaçable : même si le contexte restant est vide, la multi-abstraction subsiste et seul un destructeur $\bullet$ peut la supprimer.

[4]Cette distinction évoque certains langages, comme Python, où l'application usuelle est totale mais où une construction spécifique a été ajoutée pour gagner le confort de l'application partielle. La question importante ici est le typage statique d'un tel calcul, ce qui n'entre pas du tout en ligne de compte en Python.

La règle de sous-typage $\{\Gamma, \Delta\} \to t \leq \{\Gamma\} \to \{\Delta\} \to t$ (où Δ n'est pas effaçable) n'est plus pertinente, mais la nouvelle règle (10) qui permet de promouvoir un argument optionnel en argument obligatoire semble désormais valide : une application partielle est toujours partielle, que la fonction utilisée ait des arguments obligatoires ou optionnels.

Par exemple, la fonction $f := \lambda\{l : x\}.(x\{l' = 1\}\bullet)$ accepte toutes les fonctions attendant un entier sur l'étiquette l', que cet argument soit obligatoire ou non. L'application partielle vide, qui aurait exclu les arguments x de type $\{?l' : int\} \to int$ dans le système précédent, n'a désormais plus jamais d'effet. En donnant à notre fonction le type $\forall\alpha.\ \{l : \{l' : int\} \to \alpha\} \to \alpha$, on peut l'appliquer à $g_1 := \lambda\{l' : x\}.x$ aussi bien qu'à $g_2 := \lambda\{l' : x = 3\}.x$, car le type canonique de ce dernier terme est désormais sous-type de $\{l' : int\} \to int$. Mais on ne peut l'appliquer à $g_3 := \lambda\{l' : x, l'' : y\}.y$, à juste titre, car on ne pourrait alors pas réduire le destructeur $\bullet$.

Considérons maintenant la fonction $f' := \lambda\{l : x\}.x\{l' = 1\}$, qui peut cette fois être appliquée à g_1, à g_2 ou à g_3. Le type de f' doit indiquer que cette fonction prend en paramètre sur l n'importe quelle fonction acceptant un paramètre entier sur l', et éventuellement d'autres paramètres, et renvoie une fonction qui n'attend plus que ces autres paramètres. Pour rendre compte de ceci, il faudrait enrichir notre système de types en lui ajoutant des variables de rangée Γ représentant des ensembles de liaisons et donner à f' un type de la forme $\forall\Gamma.\forall\alpha.\ \{l : \{l' : int, \Gamma\} \to \alpha\} \to \{\Gamma\} \to \alpha$. La définition complète et l'étude d'un tel système seraient intéressantes et sont laissées pour de futurs travaux.

3. Extensions

3.1. Les contraintes de type

En pratique, Liquidsoap implémente un langage déja étendu par rapport au système décrit formellement en Section 2. On y trouve en effet une notion de *contrainte* permettant du polymorphisme *ad-hoc*. Par exemple, malgré la distinction entre entiers (de type *int*) et flottants (de type $float$), la même fonction d'addition permet d'additionner des valeurs de l'un ou l'autre des types. Celle-ci a en effet le type $\forall\alpha \in Num.\ \{\alpha, \alpha\} \to \alpha$, où la quantification universelle sur α est restreinte par une contrainte qui impose à la variable de type α d'être instanciée par *int* ou par $float$. Concrètement, le type de l'addition dans Liquidsoap est :

```
(+) :: ('a,'a)->'a where 'a is a number type
```

Outre la restriction à un type numérique, d'autres contraintes sont possibles. Par exemple, la contrainte Ord représente une classe de types naturellement ordonnables auxquels seront restreintes les opérations de comparaison : la clôture de $\{int, float, string, bool, unit\}$ par $- \times -$ (constructeur des types des paires) et $[-]$ (constructeur des types des listes). Mais Liquidsoap dispose aussi de contraintes plus exotiques, spécifiques à son domaine d'application.

Ce système très utile est néanmoins très faible. Les contraintes sont des atomes prédéfinis dans le langage et non définissables par l'utilisateur, sans autre sémantique qu'un test de satisfaction. Le système de types n'est par exemple pas capable de simplifier un ensemble de contraintes ou de détecter une incompatibilité entre contraintes.

Lors de l'inférence de type usuelle, des méta-variables sont utilisées pour représenter les types encore inconnus. Pour l'étendre aux contraintes, on attache aux méta-variables des ensembles de contraintes à respecter. Quand on instancie une méta-variable par un type, on vérifie que ses contraintes sont respectées par le type, en propageant si besoin les contraintes aux nouvelles méta-variables. Par exemple, si l'on procède à une comparaison (`x==x`) sur la variable `x`, la méta-variable de type associée (`?Tx`) est contrainte à être dans Ord ; si ensuite on utilise `x` comme une liste (`list.tl(x)==x`), la méta-variable `?Tx` est instanciée en `[?T]` (c'est-à-dire une liste dont le type est la méta-variable `?T`) et la contrainte Ord est propagée à la méta-variable `?T`.

3.2. Raffinements des types spécifiques à la génération de flux

Dans le contexte de la génération de flux audio pour des radios, il est intéressant de garantir certaines propriétés. La *vivacité* est par exemple importante : une radio doit diffuser en continu. Liquidsoap procède actuellement à la vérification qu'une source émise vers un serveur de diffusion a toujours des données à émettre. Ceci permet au concepteur de la source de s'assurer qu'il a bien prévu des dispositifs de secours dans tous les cas. Par exemple, une liste de lecture de fichiers distants n'est pas infaillible, car une panne ou une latence du réseau pourrait empêcher d'obtenir les fichiers à jouer ou encore ces fichiers pourraient être corrompus. Un opérateur de choix est vivace si l'une de ses sources filles l'est ; la vivacité sera ainsi typiquement assurée grâce à un choix par défaut qui en cas de problème va jouer un fichier ou une liste de lecture locale dont on aura préalablement vérifié les fichiers.

Cette vérification est pour le moment faite de façon dynamique : ce sont les opérateurs de sortie créés par les programmes Liquidsoap qui se chargent de demander à leurs sources filles de calculer leur vivacité. Cette approche est limitée, et ne prend notamment pas en compte les transitions. La vivacité d'un opérateur peut en effet être compromise par l'utilisation d'une fonction de transition, si celle-ci transforme une source représentant la nouvelle piste en une source qui ne produit rien[5]. Pour rejeter les programmes qui peuvent avoir ce comportement, il faudrait développer une véritable technique d'analyse statique, impliquant une annotation des types des sources.

Un autre problème est lié à la notion de temps. Au premier abord, il semble que toutes les sources évoluent dans la même échelle de temps, produisant toutes leur flux avec le même débit. Cela n'est plus vrai avec des opérateurs comme `cross`, qui superposent deux portions consécutives du même flux, accélérant localement son débit en amont. Or, on risque d'obtenir des résultats incohérents si deux opérateurs accèdent à une même source dans des échelles de temps différentes. Là encore, une solution serait d'adapter le système de types pour assurer qu'une source potentiellement utilisée (directement ou pas) par un opérateur comme `cross` n'est utilisée par aucun autre opérateur.

Enfin, Liquidsoap est actuellement restreint à l'émission de données audio. De plus, toutes les sources au sein d'une même instance partagent le même format audio (nombre de canaux et fréquence d'échantillonnage). Cependant, les abstractions fournies par notre langage ne sont pas spécifiques à la manipulation de données audio, et on pourrait par exemple espérer traiter un flux vidéo – le traitement vidéo à la volée devenant accessible aux processeurs modernes. Les formats des flux ne seraient alors pas uniformes, certains flux contenant des données audio, d'autres de la vidéo. Plus modestement, il serait utile qu'une radio puisse émettre plusieurs flux audio dont certains seraient en stéréo et d'autres en 5 canaux (*surround 5.1*). Il faudrait donc étendre le système de types pour pouvoir spécifier dans le type des sources le format du flux qu'elles émettent, ce qui permettrait de vérifier statiquement la compatibilité des formats des sources en interaction.

Conclusion

Nous avons présenté un outil puissant dédié à la génération de flux audio. Liquidsoap permet de décrire de façon simple les configurations les plus complexes, grâce à la manipulation des flux de façon abstraite.

Ce langage illustre la possibilité d'utiliser avec succès, pour des utilisateurs néophytes de la programmation, des technologies réputées compliquées comme le typage statique et inféré. Cette expérience nous a conduits à revisiter la conception des arguments étiquetés développée pour le langage OCaml et à en développer une variante plus souple. Nous espérons voir l'idée et l'implémentation réutilisées dans d'autres projets similaires.

[5]Au delà de la vivacité, c'est même un arrêt total du système qui survient dans ces cas, qui ne se produisent heureusement que très rarement et avec des programmes complexes.

Remerciements. Un grand merci à Clément Renard pour ses contributions à Liquidsoap et ses remarques sur cet article. Merci aussi à Romain Beauxis, Stéphane Gimenez et Vincent Tabard pour leurs contributions à Savonet/Liquidsoap.

Références

[1] Dolebraï. `http://www.dolebrai.net/`.

[2] RadioPi. `http://www.radiopi.org/`.

[3] H. Aït-Kaci and J. Garrigue. Label-selective λ-calculus : syntax and confluence. *Theoretical Computer Science*, 151 :353–383, 1995.

[4] C. Baccigalupo and E. Plaza. A Case-Based Song Scheduler for Group Customised Radio. *Lecture Notes in Computer Science*, 4626 :433, 2007.

[5] D. Baelde, R. Beauxis, S. Gimenez, S. Mimram, V. Tabard, and al. Savonet. `http://savonet.sf.net/`.

[6] A. Benveniste and G. Berry. The synchronous approach to reactive and real-time systems. *Proceedings of the IEEE*, 79(9) :1270–1282, 1991.

[7] E. Chailloux, P. Manoury, and B. Pagano. *Développement d'applications avec Objective Caml*. O'Reilly, 2000.

[8] L. Damas and R. Milner. Principal type schemes for functional programs. In *POPL*, 1982.

[9] J. P. Furuse and J. Garrigue. A label-selective lambda-calculus with optional arguments and its compilation method. RIMS Preprint 1041, Kyoto University, October 1995.

[10] G. Wang and P. R. Cook. ChucK : A Concurrent, On-the-fly Audio Programming Language. *Proceedings of International Computer Music Conference*, pages 219–226, 2003.

[11] William Thies, Michal Karczmarek and Saman Amarasinghe. StreamIt : A Language for Streaming Applications. *Proceedings of International Conference on Compiler Construction*, 2002.

Le caractère ‘ à la rescousse

Factorisation et réutilisation de code grâce aux variants polymorphes

Boris Yakobowski

INRIA Rocquencourt,
http://www.yakobowski.org

Résumé

Les variants polymorphes sont une fonctionnalité puissante du système de types du langage OCaml, dont l'utilisation reste pourtant rare. Dans cet article, nous nous attachons à montrer en quoi ils sont plus expressifs que les types inductifs usuels, à la fois en terme de garantie statique d'invariants et de factorisation de code. Pour cela, nous proposons trois exemples tirés d'un typeur. Aucune connaissance préalable sur les variants polymorphes n'est nécessaire.

1. Introduction

1.1. Le Graal : transformer de façon typée des types inductifs

Une application très courante des langages de la famille ML est de *raffiner* une structure inductive en une autre, plus “précise”, à travers des fonctions de traduction. Malheureusement, lorsque les deux structures sont relativement proches, le programmeur doit souvent choisir entre les garanties statiques offertes par le typage et la verbosité du code, un choix peu enviable.

Appelons en effet τ le type inductif sur lequel on travaille, et f une fonction de transformation. Une première possibilité est de donner à cette fonction le type $\tau \rightarrow \tau$. Cette approche ne reflète toutefois pas (au niveau des types) le fait que certaines constructions de τ peuvent ne plus apparaître dans les valeurs produites par f. En particulier, les fonctions analysant ces valeurs vont typiquement utiliser des assertions **assert false** pour encoder le fait qu'elles “savent” que de telles constructions sont impossibles.

La deuxième solution est de définir un second type inductif τ' encodant les invariants résultant de l'application de f. Malheureusement, avec les types inductifs usuellement disponibles dans les langages ML, certains invariants sont difficilement exprimables. De plus, les fonctions auxiliaires écrites sur τ doivent souvent être entièrement réécrites pour agir sur des valeurs de type τ'.

Une solution pour résoudre (au moins partiellement) ce problème est l'utilisation des variants polymorphes. Ces derniers permettent en effet d'encoder des invariants relativement fins, tout en permettant une très grande factorisation de code.

1.2. Introduction aux variants polymorphes

Les variants polymorphes sont une extension du langage OCaml [8] permettant d'utiliser, sans déclaration préalable, un constructeur. Celui-ci se comporte en surface comme le constructeur d'un type inductif usuel ; en particulier, il peut ou non prendre des arguments. La seule différence syntaxique est la présence d'un caractère apostrophe oblique ‘ devant le nom du constructeur.

```
# let a = `A and b1 = `B (1, true) and b2 = `B "foo"
val a : [> `A ] = `A
val b1 : [> `B of int * bool ] = `B (1, true)
val b2 : [> `B of string ] = `B "foo"
```

Dans cet exemple on vient de définir trois valeurs, a, b1 et b2, dont les valeurs sont respectivement :

1. le constructeur `A ;
2. le constructeur `B avec pour argument[1] un n-uplet de type int * bool ;
3. le constructeur `B avec pour argument la chaîne "foo".

Il est important de remarquer que b1 et b2 "partagent" le constructeur `B, et l'utilisent avec des arguments de types différents. Ceci est rendu possible par l'absence de déclaration des variants polymorphes, ainsi que par leur typage, sur lequel nous allons revenir.

Créons une liste contenant des variants polymorphes.

```
# let l1 = [ `A ; `B 1 ; `C false ];;
val l1 : [> `A | `B of int | `C of bool ] list = [`A; `B 1; `C false]
```

Le type donné à l1 est donc "liste d'éléments de type [> `A | `B of int | `C of bool]". Les crochets autour de ce type, qui sont des délimiteurs syntaxiques (et n'ont donc rien à voir avec le fait qu'on a défini une liste !), peuvent être omis.

Le type [> `A | `B of int | `C of bool ] se lit de la façon suivante : "un type comprenant au moins les constructeurs `A, `B et `C, `B prenant un argument de type int et `C de type bool". Le caractère > indique le "au moins" de notre explication, et cache en fait une construction de typage appelée *variable de rangée*. Celle-ci permet d'étendre le type donné à l1.

```
# let l2 = `A :: `D :: l1 ;;
val l2 : [> `A | `B of int | `C of bool | `D ] list = [ `A; `D; `A; `B 1; `C false]
```

On a ajouté `A et `D aux éléments présents dans l1, et le type de l2 reflète le fait qu'elle contient aussi le constructeur `D (sans argument). Nous n'entrerons pas dans les détails, mais cette extension s'est faite en instanciant la variable de rangée cachée.

Donnons d'autres exemples :

```
# (* Erreur si on ajoute un constructeur avec un type incompatible *)
let _ = `B 'b' :: l1 ;;
                  ^^
This expression has type [> `A | `B of int | `C of bool ] list
but is here used with type [> `B of char ] list
Types for tag `B are incompatible
```

La liste n'est pas hétérogène : les constructeurs doivent avoir des arguments compatibles.

```
# (* On décide que le type l1 ne peut plus être étendu *)
let l3 = (l1 : [ `A | `B of int | `C of bool ] list )
val l3 : [ `A | `B of int | `C of bool ] list = [`A; `B 1; `C false]

(* Erreur lorsqu'on essaie d'ajouter un nouveau constructeur *)
let _ = `E :: l3 ;;
            ^^
This expression has type [ `A | `B of int | `C of bool ] list
but is here used with type [> `E ] list
The first variant type does not allow tag(s) `E
```

[1] Contrairement à ceux des inductifs usuels, les constructeurs de variants polymorphes ne sont jamais "aplatis". Ici, `B prend bien en argument le n-uplet (1, true), et pas les deux arguments 1 et true.

Sans variable de rangée, le type de l3 ne peut plus être étendu (au moins par instanciation).

Terminons par la définition d'une fonction :

```
# let f = function
  | `A → `B
  | `C | `D → `D;;
val f : [< `A | `C | `D ] → [> `B | `D ] = <fun>
```

Comme on peut le constater, le filtrage sur les variants polymorphes peut s'écrire de la même façon que sur les inductifs usuels. (On verra néanmoins dans les sections suivantes une nouvelle construction, plus puissante.)

Cet exemple fait apparaître la construction duale de >, qui signifie “au plus” et se note <. La fonction f accepte en argument un type variant contenant au plus les constructeurs `A, `C et `D, et renvoie au moins les constructeurs `B et `D.

On peut instancier l'argument et le type de retour de f pour rendre ce type moins informatif.

```
# let f' = (f : ([ `A | `C ] → [`B | `D | `E ]));;
val f' : [ `A | `C ] → [ `B | `D | `E ] = <fun>
```

Étant donné que nos annotations de types ne mentionnent plus de variables de rangées, toute instanciation est devenue impossible. Toutefois, il reste possible d'utiliser l'opérateur de sous-typage :> de OCaml. Néanmoins, son utilisation doit être *explicite*, alors que l'instanciation est implicite.

```
# let f'' = (f' :> ([`A ] → [ `B | `D | `E | `F ]));;
val f'' : [ `A ] → [ `B | `D | `E | `F ] = <fun>
```

De façon générale, le type [`X | `Y] est un super-type du type [`X ]. En particulier, toute fonction acceptant un argument de type [ `X | `Y ] peut être coercée de façon à recevoir un argument de type [ `X]. En effet, le sous-typage de OCaml est structurel, et utilise la variance des constructeurs, le constructeur flèche étant covariant à droite et contravariant à gauche. Dans notre exemple nous avons réduit le type des arguments acceptés, et étendu le type de retour.

Efficacité des variants La représentation à l'exécution des inductifs usuels et des variants polymorphes a des conséquences sur la compilation des filtrages. Pour les inductifs usuels, des tables de sauts peuvent être utilisées, permettant un branchement en temps constant. A contrario, un filtrage sur n constructeurs variants polymorphes est compilé vers un arbre de sauts, de hauteur $\log(n)$. Par ailleurs, un constructeur de variants polymorphes prenant $k \geq 2$ arguments est toujours[2] représenté comme un constructeur vers un n-uplet de taille k, résultant en une représentation mémoire légèrement moins efficace. Une analyse plus poussée de ces questions se trouve dans [1, § 4].

À propos des exemples Dans la suite, nous utilisons presque systématiquement des annotations de types sur les fonctions que nous définissons, dans la mesure où ce style très déclaratif permet d'obtenir des types plus lisibles, des messages d'erreurs plus clairs, et des avertissements en cas de filtrage non exhaustif. Toutefois, ces annotations sont *toujours* facultatives, l'inférence de types sur les variants polymorphes étant totale.

1.3. Plan

Cet article comporte trois parties, qui introduisent chacunes un exemple (de complexité approximativement croissante). Nous expliquons certaines nouvelles constructions liées aux variants

[2]Cette restriction ne résulte pas d'une impossibilité théorique, mais provient de l'ambiguïté existant dans la syntaxe concrète de OCaml entre un constructeur prenant 2 arguments et un constructeur prenant un n-uplet de 2 éléments.

polymorphes au fur et à mesure de leur introduction ; aussi, et bien que les exemples soient totalement indépendants, une lecture dans l'ordre est conseillée.

La section 2 examine le problème de la résolution des constructeurs de types par un typeur. La section 3 montre comment garantir par typage que certaines valeurs sont en forme normale (pour une relation donnée), à travers l'exemple des types d'un langage avec quantification bornée. La section 4 montre comment "désucrer" de façon typée l'arbre de syntaxe abstraite issue d'une phase d'analyse syntaxique.

Le code source des exemples proposés n'est pas toujours complet, par manque de place. La version intégrale peut être trouvée à l'adresse `http://www.yakobowski.org/jfla08.html`

2. Constructeurs de types non résolus

L'une des tâches d'un typeur est la *résolution de types*. Considérons en effet la déclaration **let** id (x : t) = x. Lorsqu'il rencontre le type t, le typeur doit le résoudre, c'est-à-dire trouver quelle est son identité exacte. En effet, t peut avoir été déclaré dans deux modules ouverts, ou même déclaré deux fois dans le module courant (*e.g.* **type** t = A | B;; **type** t = C | D;;). L'*identité* d'un type doit donc être une information plus précise que son simple nom ; on utilise typiquement un couple comprenant le nom et une valeur unique issue d'un compteur.

Pour fixer les idées, on définit les *constructeurs de types*[3] et les *types* par les grammaires suivantes :

$$C \quad := \quad \texttt{int} \mid \texttt{float} \mid \rightarrow \mid ()^n \mid I^i \qquad\qquad \tau \quad := \quad \alpha \mid C\,\overline{\tau}$$

Le symbole $()^n$ représente le constructeur des n-uplets de longueur n, tandis que I^i représente l'inductif dont le nom est I et l'identité i. Les types de ML sont alors définis inductivement comme étant soit une variable de type α, soit l'application d'un constructeur de types à un ou plusieurs types.

2.1. Des solutions en ML

Les deux grammaires données ci-dessus se traduisent très naturellement en ML :

```
type typeconstr = Int | Float | Arrow | Uple of int | Inductive of inductive_id
type ml_type = Var of var | Constr of typeconstr * ml_type list
```

Nous laissons le type des variables de types non spécifié car il n'a pas d'importance ici. De même, l'identité d'un type inductif n'a pas besoin d'être précisée.

Un inductif pour l'analyse syntaxique Il est difficile, au moment de l'analyse syntaxique, de connaître l'identité des inductifs ; de plus, cela mélangerait typage et analyse syntaxique, une approche qui n'est pas encouragée. La solution traditionnellement retenue est de définir un arbre de syntaxe abstraite spécifique à l'analyse syntaxique, distinct de celui utilisé lors du typage. Le constructeur correspondant aux types inductifs a alors comme paramètre une chaîne de caractères (le nom du type) au lieu d'une identité. On définit également un second type inductif pour les types ML.

```
type parsing_typeconstr = PInt | PFloat | PArrow | PUple of int | PInductive of string
type parsing_ml_type = PVar of var | PConstr of parsing_typeconstr * parsing_ml_type list
```

Étant données ces définitions, une fonction résolvant les constructeurs de types prend en argument un environnement de typage et un objet de type parsing_ml_type, et renvoie un objet de type ml_type (ou lève une exception si un inductif non préalablement défini est rencontré).

[3] D'autres constructeurs de types tels que char, array, ... peuvent bien sûr être ajoutés.

Dans notre algèbre de types simplifiée, la redondance entre les deux définitions est flagrante. Mais elle apparaît également dans des systèmes plus complexes : dans le compilateur OCaml, les types Parsetree.core_type_desc et Types.type_desc ont 10 et 12 constructeurs, et en partagent 7.

Évidemment, cette redondance a un coût. Il faut par exemple écrire un afficheur pour chacun des deux types, avec peu de partage de code possible. De même la fonction résolvant un constructeur de types est composée principalement d'un fastidieux filtrage.

```
let resolve_typeconstr env = function
  | PInt → Int | PFloat → Float | PArrow → Arrow | PUple n → Uple n
  | PInductive s → Inductive (resolve_inductive env s)
```

Ici, les 4 premiers cas sont évidents ; néanmoins, étant données les définitions de typeconstr et parsing_typeconstr, il n'existe pas de solution respectant la discipline de types de ML permettant de les factoriser.

D'un point de vue compilation, le résultat est mitigé. Si les constructeurs des deux types sont donnés *exactement* dans le même ordre, la représentation des inductifs à l'exécution (qui est non typée) permet que les 3 premiers cas de filtrage soient traduits par des opérations *nop*. En revanche, la traductions PUple n → Uple n provoque systématiquement l'allocation d'un nouveau bloc, bien que les deux blocs aient en fait la même représentation mémoire.

Un inductif pour les cas communs Une évolution naturelle est de factoriser dans un troisième type les déclarations communes aux deux inductifs (*i.e.* tous les constructeurs autres que Inductive et PInductive). La fonction resolve_constr devient alors beaucoup moins verbeuse.

```
type common_typeconstr = Int | Float | Arrow | Uple of int
type typeconstr = Common of typeconstr | Inductive of inductive_id
type parsing_typeconstr = PCommon of typeconstr | PInductive of string

let resolve_typeconstr env = function
  | PCommon c → Common c
  | PInductive s → Inductive (resolve_inductive env s)
```

Cette solution est a priori plus séduisante. Toutefois, elle a pour inconvénient d'introduire deux constructeurs (Common et PCommon) qui n'ont sémantiquement aucune raison d'être. Par ailleurs, cette stratification a un coût à l'exécution : tous les constructeurs de types ne correspondant pas aux inductifs doivent subir une indirection. La simplicité d'écriture s'obtient au détriment de l'efficacité et de l'intelligibilité.

Paramétrer typeconstr Une troisième possibilité est de paramétrer typeconstr par le type du constructeur Inductive.

```
type 'a typeconstr_aux = Int | Float | Arrow | Uple of int | Inductive of 'a
type typeconstr = inductive_id typeconstr_aux
type typeconstr_parsing = string typeconstr_aux
```

Cette solution a plusieurs défauts. Tout d'abord, elle ne passe pas vraiment à l'échelle. En effet, elle requiert un paramètre par constructeur prenant des arguments différents lors de l'analyse syntaxique et lors des phases ultérieures. (Dans notre cas, on pourrait par exemple rajouter un constructeur Alias pour les alias de types.) Ensuite, le type typeconstr_aux est finalement “trop” polymorphe : quel serait en effet le sens d'une fonction prenant un argument de type float typeconstr_aux ? Enfin, sans annotation explicite, l'inférence donne souvent aux fonctions des types utilisant typeconstr_aux, peu lisibles. Ici, on veut du polymorphisme *ad hoc* plutôt que du polymorphisme paramétrique.

2.2. Avec des variants polymorphes

Les variants polymorphes permettent de marier harmonieusement les deux premières solutions, en évitant les écueils de la troisième. En effet, ils autorisent le partage des constructeurs communs entre les types typeconstr et typeconstr_parsing.

Nous commençons par mettre en évidence ces constructeurs dans un type common_typeconstr. En dehors des crochets de délimitation et du caractère ‘ précédant les constructeurs, cette déclaration et celle de la solution 2 sont identiques.

```
type common_typeconstr = [ `Int | `Float | `Arrow | `Uple of int ]
```

L'apport d'expressivité apparaît dans la définition de typeconstr et parsing_typeconstr. Il devient en effet possible *d'étendre* la définition ci-dessus, en ajoutant un nouveau constructeur. Notons que l'on a choisi ici d'utiliser le (même) nom `Inductive dans les deux déclarations.

```
type typeconstr = [ common_typeconstr | `Inductive of inductive_id ]
type parsing_typeconstr = [ common_typeconstr | `Inductive of string ]
```

Il est intéressant d'étudier la réponse du typeur OCaml sur ces déclarations.

```
# type typeconstr = [ common_typeconstr | `Inductive of inductive_id ];;
type typeconstr = [ `Arrow | `Float | `Inductive of inductive_id | `Int | `Uple of int ]
# type parsing_typeconstr = [ common_typeconstr | `Inductive of string ];;
type parsing_typeconstr = [ `Arrow | `Float | `Inductive of string | `Int | `Uple of int ]
```

Tout se passe exactement comme si on avait expansé common_typeconstr lors de la définition de typeconstr et parsing_typeconstr, ces deux types semblant “partager” les constructeurs de common_typeconstr. Une telle vision est toutefois trompeuse : comme on l'a vu dans la section 1.2, ces constructeurs “préexistent” et peuvent parfaitement être utilisés en dehors des types qu'on vient de définir. En fait, typeconstr et parsing_typeconstr ne sont rien de plus que des *alias* de types.

Il nous reste à voir comment écrire des fonctions d'affichage, ainsi que resolve_typeconstr. L'écriture d'un afficheur pour common_typeconstr se fait exactement de la même façon qu'avec des inductifs standards (modulo l'ajout des caractères ‘), et de façon très similaire pour typeconstr_parsing.

```
let print_typeconstr_common = function
  | `Int → print_string "int"
  | _ → ...

let print_typeconstr_parsing = function
  | #common_typeconstr as x → print_typeconstr_common x
  | `Inductive s → print_string s
```

La construction #common_typeconstr discrimine en fonction des constructeurs présents dans common_typeconstr : toutes les valeurs commençant par l'un de ces constructeurs (`Int, `Float, `Arrow ou `Uple) sont traitées par la première branche du filtrage. Si au contraire la valeur est de la forme `Inductive (qui n'est pas un constructeur de common_typeconstr), elle est traitée par le deuxième cas. Cet opérateur permet d'effectuer l'opération PCommon c → Common c de la solution 2 de la section précédente, sans marquer les valeurs par les constructeurs Common ou PCommon.

Le type inféré pour la fonction print_typeconstr_parsing est :

```
[< `Arrow | `Float | `Inductive of string | `Int | `Uple of int ] → unit
```

On reconnaît la déclaration expansée de parsing_typeconstr, modulo le caractère <.

Comme le montre la fonction ci-dessous, il est possible (et utile) de contraindre la signature de la fonction lors de sa définition afin d'obtenir des types plus courts (et donc plus lisibles).

```
# let resolve_typeconstr env : parsing_typeconstr → typeconstr = function
  | #common_typeconstr as x → x
  | `Inductive s → `Inductive (resolve_inductive env s)

val resolve_typeconstr : env → parsing_typeconstr → typeconstr
```

Supposons que resolve_typeconstr reçoive comme argument la valeur `Inductive "tree". Si le type tree est dans l'environnement, la fonction va renvoyer la valeur `Inductive id_tree, où id_tree est l'identité du constructeur de types tree. Ainsi, le constructeur `Inductive est utilisé avec deux arguments de types incompatibles, tout cela de manière sûre (et sans qu'il ait été besoin de paramétrer typeconstr par le type de l'argument de `Inductive, comme dans la troisième solution de la Section 2.1).

3. Sous-ensembles garantis statiquement

Le langage ML^F [7] est une extension conservative de ML et du Système F. Dans cette section, nous allons nous intéresser à ses types, et montrer qu'on peut facilement, en utilisant des variants polymorphes, caractériser un ensemble de ces types contenant tous ceux en forme normale. Cet exemple est un peu inhabituel, mais se généralise facilement à de nombreuses fonctions transformant une structure inductive.

3.1. Les types de ML^F

Les types de ML^F sont stratifiés en d'un côté des *monotypes* τ, et de l'autre des *polytypes* σ. Les grammaires suivantes décrivent ces deux classes syntaxiques.

$$\tau \quad ::= \quad \alpha \mid C\,\overline{\tau} \qquad\qquad \sigma \quad ::= \quad \tau \mid \bot \mid \forall\,(\alpha \diamond \sigma)\;\sigma$$

Les monotypes correspondent exactement aux types de ML. Les polytypes sont soit des monotypes, soit le type $\bot$ (qui correspond au type du système F $\forall\alpha.\,\alpha$), soit une forme de quantification bornée $\forall\,(\alpha \diamond \sigma)\;\sigma'$ se lisant très approximativement « σ' dans lequel la variable α parcourt[4] l'ensemble des types décrits par σ » ; α est lié dans σ' mais pas dans σ. On abrège $\forall\,(\alpha \diamond \bot)$ en $\forall\,(\alpha)$.

Cette grammaire se traduit très naturellement, à la fois avec et sans variants polymorphes :

```
type monotype = Var of var | Constr  of typeconstr * monotype list
type polytype = Mono of monotype | Bottom | Bound of (var * polytype) * polytype

type monotype = [ `Var of var  |  `Constr of typeconstr * monotype list ]
type polytype = [ monotype | `Bottom | `Bound of (var * polytype) * polytype ]
```

Toutefois, ces derniers capturent directement l'inclusion entre polytypes et monotypes, et évitent l'indirection introduite par le constructeur Mono. Dans la suite, on ne considérera plus que cette seconde définition.

La recherche des variables libres s'écrit sans aucune surprise[5], modulo l'utilisation de l'opérateur # pour séparer les monotypes des autres constructeurs des polytypes.

```
let rec ftv_monotype : monotype → _ = function
  | `Var v → [v]
  | `Constr (_, l) → List.concat (List.map ftv_monotype l)
```

[4] En réalité, $\diamond$ est une métavariable valant soit $\geq$, soit $=$. Toutefois cette distinction n'a pas d'importance pour l'exemple que nous allons présenter ici.

[5] Notre fonction renvoie des listes afin de simplifier les exemples. Une implémentation réaliste utiliserait des ensembles.

```
and ftv_polytype : polytype → _ = function
  | #monotype as m → ftv_monotype m
  | `Bottom → []
  | `Bound ((v, t), t') → ftv_polytype t @ (list_remove_all v (ftv_polytype t'))
```

Afin d'introduire la suite de notre exemple, nous allons stratifier la définition des polytypes en trois types distincts, les deux premiers encodant le type de $\bot$ et de la quantification bornée. Étant donné que (du point de vue du typage) les définitions de variants polymorphes sont expansées, nos deux définitions de polytype sont parfaitement équivalentes. Nous introduisons également un alias pour les monotypes qui ne sont pas des variables

```
type bot = [ `Bottom ]
type ('a, 'b) bound = [ `Bound of (var * 'a) * 'b ]
type polytype = [ monotype | bot | (polytype, polytype) bound ]

type constr = [ `Constr of typeconstr * monotype list ]
```

3.2. Équivalence entre types de ML^F

ML^F définit sur ses types une relation d'équivalence riche, qui permet notamment d'éliminer les quantifications inutiles. Une version simplifiée des règles les plus importantes est présentée ci-dessous. Les règles sont congruentes, et peuvent donc s'appliquer à l'intérieur d'un type. Étant donné un polytype σ, $\mathsf{ftv}(\sigma)$ représente ses variables libres.

$$\frac{\alpha \notin \mathsf{ftv}(\sigma')}{\forall\,(\alpha \diamond \sigma)\ \sigma' \equiv \sigma'}\ \textsc{Eq-Free} \qquad \textsc{Eq-Var}\quad \forall\,(\alpha \diamond \sigma)\ \alpha \equiv \sigma \qquad \textsc{Eq-Mono}\quad \forall\,(\alpha \diamond \tau)\ \sigma \equiv \sigma[\tau/\alpha]$$

La règle Eq-Free élimine les bornes non utilisées. La règle Eq-Var supprime les quantifications qui ne sont en fait que des alias. Enfin, Eq-Mono exprime le fait que les quantifications sur des monotypes peuvent être expansées ; l'opération de substitution est supposée éviter les captures de variables.

Il est possible de définir une forme normale pour les types. Celle-ci est obtenue en appliquant répétitivement Eq-Free, Eq-Var et Eq-Mono de la gauche vers la droite.

Donnons quelques exemples :

- $\bot$ est une forme normale, par exemple celle du type $\forall\,(\alpha \diamond \sigma)\ \bot$.
- La forme normale du type $\forall\,(\beta \diamond \alpha \to \alpha)\ \forall\,(\gamma \diamond \forall\,(\delta)\ \forall\,(\epsilon)\ \epsilon \to \beta)\ \beta \to \gamma$ est le type $\forall\,(\gamma \diamond \forall\,(\epsilon)\ \epsilon \to (\alpha \to \alpha))\ (\alpha \to \alpha) \to \gamma$. En effet, la variable δ est inutilisée et peut être supprimée par Eq-Free, tandis que la variable β peut être expansée grâce à Eq-Mono.

3.3. Sous-ensembles et sous-types

En étudiant attentivement les règles d'équivalence, on se rend compte que les polytypes en forme normale sont contenus dans le sous-ensemble σ_n de σ décrit par la grammaire suivante :

$$\begin{array}{rcl} \sigma_n & ::= & \tau \mid \bot \mid \forall\,(\alpha \diamond \sigma_\bot)\ \sigma_\tau \\ \sigma_\bot & ::= & \bot \mid \forall\,(\alpha \diamond \sigma_\bot)\ \sigma_\tau \\ \sigma_\tau & ::= & C\,\overline{\tau} \mid \forall\,(\alpha \diamond \sigma_\bot)\ \sigma_\tau \end{array}$$

Il est possible (et aisé) de décrire ces trois ensembles en utilisant les variants polymorphes. A contrario, encoder une telle grammaire avec des inductifs usuels est lourd ; il faut 3 constructeurs pour σ_n, 2 constructeurs pour σ_τ et 2 pour $\sigma_\bot$, ainsi que des fonctions de conversion.

```
type nf_polytype = [ monotype | bot | (bot_poly, constr_poly) bound ]
and bot_poly  =    [            bot | (bot_poly, constr_poly) bound ]
and constr_poly =  [ constr     |     (bot_poly, constr_poly) bound ]
```

(Malheureusement il ne semble pas possible de factoriser (bot_poly, constr_poly) bound dans un type séparé afin d'éviter la légère duplication de code, le typeur de OCaml rejetant une telle déclaration.)

Comme on l'a vu, σ_n est un sous-ensemble de σ. Utiliser des de variants polymorphes permet de transposer cette relation au niveau des types : nf_polytype est un *sous-type* de polytype ! En pratique, cela veut dire que toute valeur de type nf_polytype peut être *coercée* en une valeur de type polytype, comme le montre la fonction de conversion suivante :

```
let nf_to_polytype t = (t : nf_polytype :> polytype)
```

Cette fonction a pour type nf_polytype → polytype, et convertit donc un polytype en forme normale en un polytype générique. *Cette conversion est purement logique* : l'opérateur de sous-typage :> de OCaml n'a aucun coût à l'exécution, et nf_to_polytype est compilée vers la fonction identité ! C'est là un gain crucial par rapport à une approche utilisant les inductifs usuels, qui aurait demandée une traversée et une reconstruction de tout le terme représentant t.

Fonctions de substitutions Nous commençons par écrire les fonctions substituant un monotype dans les monotypes et les polytypes. Dans ce second cas on suppose que les variables libres du monotype sont différentes des variables liées dans le polytype, afin d'éviter toute capture de variable.

```
# let rec expand_mono_in_mono (v, m) = function
  | `Var v' → if v = v' then m else `Var v'
  | `Constr (c, l) → `Constr (c, List.map (expand_mono_in_mono (v, m)) l)
and expand_mono_in_poly (v, m) : polytype → _ = function
  | #monotype as m' → (expand_mono_in_mono (v, m) m' :> polytype)
  | `Bottom → `Bottom
  | `Bound ((v', t), t') → `Bound ((v', expand_mono_in_poly (v, m) t),
                                   if v = v' then t' else expand_mono_in_poly (v, m) t')

val expand_mono_in_mono : var * monotype → monotype → monotype = <fun>
val expand_mono_in_poly : var * monotype → polytype → polytype = <fun>
```

L'annotation de types sur expand_mono_in_poly est facultative. En revanche, la coercion expand_mono_in_mono (v, m) m' :> polytype est essentielle. En effet, expand_mono_in_mono renvoyant un monotype, en l'absence de coercion la première branche du filtrage imposerait aux autres branches de renvoyer également un monotype. Ceci est contradictoire avec le fait que la fonction peut par exemple renvoyer `Bottom.

Normalisation La fonction de normalisation s'écrit très naturellement. Si le type est un monotype ou ⊥, le résultat est immédiat. Sinon, c'est une quantification bornée, et on normalise successivement ses deux sous-parties, en utilisant au vol toute règle applicable. On suppose qu'aucune variable n'est liée deux fois, et que les variables libres et liées sont disjointes.

```
let rec nf : polytype → nf_polytype = function
  | #bot | #monotype as t → t
  | `Bound ((v, t), t') →
      match nf t with
        | #monotype as m →  nf (expand_mono_in_poly (v, m) t') (* Eq−Mono *)
        | #bot_poly as t →
```

```
        match nf t' with
          | `Var v' → if v = v' then t (* Eq−Var *) else `Var v' (* Eq−Free *)
          | #bot → `Bottom (* Eq−Free *)
          | #constr_poly as t' →
              let ftv = ftv_polytype (t' : constr_poly :> polytype) in
              if List.mem v ftv then `Bound ((v, t), t')
              else t' (* Eq−Free *)
```

Comme précédemment, l'annotation `polytype → nf_polytype` est facultative. En revanche, la coercion `(t' : constr_poly :>polytype)` est nécessaire. En effet, dans ce contexte, `t'` a pour type `constr_poly`, qui n'est pas compatible avec `polytype` (qui est le type des arguments de `ftv_polytype`). Sans coercion, on obtient le message d'erreur suivant :

```
1  This expression has type
2    [> constr_poly ] =
3      [> `Bound of (var * bot_poly) * constr_poly
4       | `Constr of constr * monotype list
5       | `Var of var ]
6  but is here used with type polytype =
7      [ `Bottom
8      | `Bound of (var * polytype) * polytype
9      | `Constr of constr * monotype list
10     | `Var of var ]
11 Type bot_poly = [ `Bottom | `Bound of (var * bot_poly) * constr_poly ]
12 is not compatible with type polytype =
13     [ `Bottom
14     | `Bound of (var * polytype) * polytype
15     | `Constr of constr * monotype list
16     | `Var of var ]
17 The first variant type does not allow tag(s) `Constr, `Var
```

Détaillons : `t'` a le type `[> constr_poly]`, qui n'est pas compatible avec le type `polytype` (lignes 1–10). En effet, les arguments du constructeur `` `Bound `` dans les deux types (lignes 3 et 8) sont incompatibles : les types `bot_poly` et `polytype` sont incompatibles (lignes 11–16), la raison étant donnée ligne 17.

Sans variants polymorphes Les inductifs traditionnels permettent, au prix d'une certaine lourdeur, d'obtenir le même niveau de garanties statiques que celui obtenu ici. L'auteur a testé cette approche. La fonction `nf` résultante fait 22 lignes, à comparer avec les 15 de la fonction ci-dessus (soit une augmentation de 46%, sans compter deux fonctions de coercion de 3 lignes chacune) ; elle est également beaucoup moins claire, car elle nécessite l'emploi de deux sous-fonctions auxiliaires.

Par ailleurs, pour obtenir la même complexité à l'exécution, il est impératif d'écrire la fonction `ftv_polytype` pour qu'elle agisse sur des valeurs de type `constr_poly` (et donc de l'écrire deux fois). Une solution utilisant une coercion `constr_poly` ⇒ `polytype` augmente en effet la complexité d'un facteur $|\tau|$ (où τ est le type coercé), la coercion devant traverser tout le type pour le convertir.

Qu'avons-nous garanti ? Le sous-ensemble σ_n que nous avons caractérisé n'est pas exactement le sous-ensemble des formes normales des types — seulement un sur-ensemble. En effet, l'application de la règle Eq-Free n'est pas capturée par le critère purement syntaxique que nous avons exprimé. Il faudrait probablement pour cela un système avec types dépendants. De façon générale, les variants polymorphes permettent d'encoder facilement tous les invariants expressibles par une grammaire BNF. En revanche, ils n'offrent pas de réelle garantie pour tout ce qui n'est pas expressible à l'aide de ce formalisme, typiquement les questions de lieurs.

4. D'un arbre à l'autre

Dans cette partie, nous allons nous intéresser au "désucrage" d'un arbre de syntaxe abstraite (AST) pour l'analyse syntaxique en un AST correspondant à un langage noyau. Ces deux langages vont partager des constructeurs, et nous allons voir comment écrire un afficheur pour les deux types correspondants.

4.1. Définition des arbres de syntaxe abstraite

Nous partons d'un langage ML usuel, avec plusieurs constructions de haut niveau telles que des opérations arithmétiques, des conditionnelles, des abstractions et applications multiples...

Notre langage cible est un langage noyau comprenant les constructions du lambda-calcul enrichi par les n-uplets, les let, et des constantes. Ce langage est celui utilisé pour le typage (et non pour la compilation, ce qui explique certains de nos choix).

e	::= x	*Variables*	
	| $e\ \overline{e}$	*Application multiples*	
	| $\lambda(\overline{x})\ e$	*Abstractions multiples*	
	| let rec$^?$ $x = e$ in e	*Let (récursif ou non)*	
	| $i \mid b$	*Entiers, Booléens*	
	| $e + e \mid e = e \mid \ldots$	*Opérations arithmétiques*	
	| if e then e else e	*Conditionnelles*	
	| $(e, ..., e)$	*N-Uplets*	

ϵ	::= x	*Variables*	
	| $\epsilon\ \epsilon$	*Application*	
	| $\lambda(x)\ \epsilon$	*Abstraction*	
	| let $x = \epsilon$ in ϵ	*Let non récursif*	
	| c	*Constantes*	
	| $(\epsilon, ..., \epsilon)$	*n-uplets*	

Les deux langages partagent de façon "exacte" deux constructions, les variables et les n-uplets. D'autres constructions sont "partiellement" partagées : les let, qui peuvent être récursifs dans le langage source et qui sont toujours non récursifs dans le langage cible, et les constantes, qui sont plus nombreuses dans le langage cible. Elles incluent en effet les entiers et les booléens, mais aussi les opérateurs arithmétiques tels que plus (de type int → int → int), un encodage des conditionnelles (de type $\forall\alpha.\mathsf{bool} \to \alpha \to \alpha \to \alpha$), ...

4.2. Récursion et variants polymorphes

On commence par définir les types des valeurs et des constructeurs de types, en les stratifiant afin de refléter le plus grand nombre de constantes présentes dans l'AST cible.

```
(* Valeurs de base *)
type ground_ct = [ `Int of int | `Bool of bool ]
(* Toutes les constantes *)
type ct = [ ground_ct | `IfTE | `OpPlus | `OpEq | `OpFix ]

(* Constructeurs de types pour les valeurs de base puis les constantes *)
type ground_typeconstr = [ `TInt | `TBool ]
type typeconstr = [ `TVar of var | ground_typeconstr | `TArrow of typeconstr * typeconstr ]
```

On introduit ensuite des déclarations auxiliaires pour les constructions let, encodant le fait qu'elles peuvent être ou non récursives. Le constructeur let_aux est paramétré par le type des expressions.

```
type nonrec = [ `NonRec ]
type recnonrec = [ `Rec | nonrec ]
type ('rec_flag, 'expr) let_aux = [ `Let of 'rec_flag * (var * 'expr) * 'expr ]
```

Dans cet exemple, la difficulté principale provient du fait que le type des expressions est récursif, ce qui complique la factorisation de code au niveau de la déclaration des types. La solution est de définir plusieurs sous-types ouverts (*i.e.* paramétrés par le type des expressions), comprenant *certains* des constructeurs. Dans un deuxième temps, on ferme la récursion (au niveau des types) en fusionnant les sous-types ainsi définis.

```
(* Constructeurs communs aux deux AST *)
type 'expr common = [ `Var of var | `Uple of 'expr list ]

(* Constructeurs dans un seul des AST, ou exactement partagés *)
type 'expr source_aux = [
  | 'expr common
  | `SeqApp of 'expr * 'expr list
  | `SeqAbs of var list * 'expr
  | `Plus of 'expr * 'expr | `Eq of 'expr * 'expr
  | `If of 'expr * 'expr * 'expr ]
type 'expr dest_aux = [
  | 'expr common
  | `App of 'expr * 'expr
  | `Abs of var * 'expr ]

(* Les deux AST, obtenus en fermant la récursion *)
type source = [ source source_aux
              | (recnonrec, source) let_aux
              | `Ct of ground_ct * ground_typeconstr ]
type dest =  [ dest dest_aux
              | (nonrec, dest) let_aux
              | `Ct of ct * typeconstr ]
```

La fonction du désucrage est récursive ; les applications multiples utilisent toutefois une fonction auxiliaire. Les constantes doivent être coercées vers le type des constantes du langage destination. Les opérateurs arithmétiques et la construction if sont traduits vers l'application de l'opérateur approprié à chaque cas. À titre d'exemple, la valeur ct_plus vaut (`OpPlus, `TArrow (`TInt, `TArrow (`TInt, `TInt))). Les let récursifs utilisent l'opérateur fix pour encoder la récursion.

```
let rec desugar : source → dest = function
  | `Var _ as v → v
  | `Uple l → `Uple (List.map desugar l)
  | `SeqApp (e, l) → desugar_seq_app (desugar e) l
  | `SeqAbs ([], e) → desugar e
  | `SeqAbs (v :: q, e) → `Abs (v, desugar (`SeqAbs (q, e)))
  | `Ct (c, t) → `Ct ((c :> ct), (t :> typeconstr))
  | `If (c, e1, e2) → desugar_seq_app (`Ct ct_if) [c;e1;e2]
  | `Plus (e1, e2) → desugar_seq_app (`Ct ct_plus) [e1;e2]
  | `Eq (e1, e2) → desugar_seq_app (`Ct ct_eq) [e1;e2]
  | `Let (`NonRec, (v, e1), e2) → `Let (`NonRec, (v, desugar e1), desugar e2)
  | `Let (`Rec, (v, e1), e2) →
      `Let (`NonRec, (v, `App (`Ct ct_fix, `Abs (v, desugar e1))), desugar e2)
and desugar_seq_app (e : dest) (* : source list → dest *) = function
  | [] → e
  | e' :: q → desugar_seq_app (`App (e, desugar e')) q
```

4.3. Le meilleur des deux arbres

Écrire un afficheur pour chacun des deux AST peut se faire de trois façons, ordonnées ici par factorisation de code croissante :

1. Écrire deux afficheurs distincts, un pour chaque type, sans factoriser de code. Ici, le code pour les n-uplets, les constantes, les let ou les variables est dupliqué.

   C'est la solution que l'on aurait obtenue si des types inductifs usuels avaient été utilisés. En effet, les deux arbres partageant peu de cas (contrairement aux exemples de la Section 2), il est naturel de définir deux types inductifs séparés.

2. Écrire un afficheur ouvert (prenant en argument un autre afficheur) pour les types common, source_aux et dest_aux, puis écrire un afficheur pour source et dest qui ferme la récursion. Le schéma est exactement le même que celui suivi pour la définition des types, comme le montre l'exemple partiel ci-dessous[6].

```
let print_common print : _ common → unit = function
  | ‘Var v → print_var v
  | ‘Uple l → print_string "(" ; print_list ", " print l; print_string ")"

let print_dest_aux print : _ dest_aux → _ = function
  | #common as e → print_common print e
  | ‘Abs (v, e) → print_string "fun "; print_var v; print_string " → ";
                  print e  (* Appel à l'afficheur passé en argument *)
  | ‘App (e, e') → print e ; print e' (* Idem *)

let rec print_dest : dest → _ = function
  | #dest_aux as e → print_dest_aux print_dest e (* Récursion *)
  | #let_aux as l → print_let print_dest l
  | ‘Ct (c, _) → [...]
```

   Cette approche permet d'obtenir du code réellement extensible.

3. La dernière possibilité, la plus adaptée ici, consiste à définir le plus petit supertype des deux AST, et à écrire un afficheur pour ce type particulier. De cette façon, le partage de code est optimal (en particulier aucune fonction auxiliaire n'est nécessaire). Les deux AST étant des sous-types de ce supertype, on obtient “gratuitement” les fonctions de conversion. De plus, grâce à la façon dont nous avons structuré nos types, l'écriture du supertype est immédiate.

```
type all_expr = [  (* Sur-type des deux AST *)
  | all_expr source_aux
  | all_expr dest_aux
  | (recnonrec, all_expr) let_aux
  | ‘Ct of ct * typeconstr ]

let coerce_source x = (x : source :> all_expr)
let coerce_dest   x = (x : dest :> all_expr)
```

   On notera au passage que le sous-typage de OCaml se fait en profondeur, comme le montre le constructeur ‘Let qui prend comme premier argument un objet de type [‘NonRec] dans dest et un objet de type [‘Rec | ‘NonRec] dans all_expr.

   L'écriture de l'afficheur est extrêmement naturelle en utilisant cette méthode. En particulier il n'est pas nécessaire d'écrire des afficheurs ouverts, la récursion pouvant être fermée directement.

[6] Nous ne traitons pas ici le problème consistant à reparenthèser les expressions, qui est orthogonal.

```
let rec print_all : all_expr → unit = function
  | `App (e, e') → print_all e ; print_all e'
(* [...] Tous les autres cas *)

let print_source e = print_all (coerce_source e)
let print_dest e = print_all (coerce_dest e)
```

Dans le cas général, la solution 2 est toujours applicable et est à notre avis toujours préférable à la solution 1. La solution 3 ne s'applique que si le supertype a toujours un "sens". C'est le cas ici, dans la mesure où l'on peut facilement mélanger les constructions des deux AST. Elle est plus brève, mais moins extensible si d'autres variantes de l'AST sont écrites.

La solution 3 impose toutefois un peu de rigueur lors de la définition des AST intermédiaires. En effet, du fait que `Ct et `Let prennent des arguments différents dans les deux AST, si on ajoute ces constructeurs directement dans les types source_aux et dest_aux, il n'est pas possible d'obtenir le supertype automatiquement en fermant la récursion (il faut réécrire tout le type "à la main"). C'est la raison pour laquelle on les a "sortis" de ces définitions dans la section 4.2.

Conclusion

Les variants polymorphes au quotidien L'auteur emploie très régulièrement des variants polymorphes dans ses développements. D'après son expérience, les difficultés rencontrées sont de trois sortes :

1. La complexité des types inférés (en particulier, ceux-ci sont souvent récursifs).
2. La longueur des messages d'erreurs lorsqu'une expression n'est pas typable.
3. Faut-il mettre une coercion à un endroit précis du code ?

En pratique, l'ajout d'annotations de types sur les fonctions récursives fait quasiment disparaître la première difficulté, et atténue la deuxième ; l'exemple de la section 3.3 montre qu'un message d'erreur sur des abréviations de types est tout à fait intelligible. Pour le troisième point, le programmeur doit avoir à l'esprit les types des fonctions sur lesquelles il travaille, et insérer une coercion dès qu'une fonction reçoit une valeur qui a un type plus contraint que le type attendu. Toutefois, si les fonctions sont annotées, le moteur d'inférence signale généralement une erreur au "bon" endroit.

Travaux connexes Jacques Garrigue [1] présente la sémantique et le schéma de compilation des variants polymorphes dans OCaml ; la section 3 donne des exemples de leur utilisation. Les exemples proposés dans cet article correspondent à la section 3.2 "Polymorphism and subtyping". D'autres exemples, qualifiés de "monomorphes" (section 3.1) sont utilisés par exemple dans la bibliothèque Lablgtk [5]. Dans un autre article [2], Garrigue décrit une solution au problème de l'extension, à travers l'exemple d'un évaluateur de λ-calcul avec constantes. Nous avons utilisé la même approche pour le point 2 de la section 4.3. Dans un troisième article [4], Garrigue décrit le typage du filtrage en présence de variants polymorphes. La spécification formelle du typage des variants polymorphes dans OCaml peut être trouvée dans [3]. Kagawa [6] propose une implémentation des variants polymorphes au-dessus du système de types de Haskell.

Conclusion L'auteur espère que cet article contribuera à démystifier les variants polymorphes, à la fois au niveau de la difficulté de leur utilisation, et de leur intérêt. Nous voudrions insister sur le fait que l'écriture de fonctions telles que nf (section 3.3), desugar (section 4.2) ou print_all (section 4.3) est très naturelle. En pratique, il est même *plus* facile d'écrire certaines fonctions (telles que nf), le système de types garantissant les invariants utilisés ; avec une approche plus traditionnelle le programmeur doit avoir en tête les cas impossibles, et les éliminer manuellement en utilisant des assertions.

Remerciements L'auteur tient à remercier Jacques Garrigue pour ses remarques et explications, Zaynah Dargaye et Benoit Razet pour leurs suggestions et leur relecture attentive, et les relecteurs anonymes pour leurs commentaires et idées.

Références

[1] Jacques Garrigue. Programming with polymorphic variants. In *ML Workshop*, Baltimore, September 1998.

[2] Jacques Garrigue. Code reuse through polymorphic variants. In *Workshop on Foundations of Software Engineering*, Sasaguri, Japan, November 2000.

[3] Jacques Garrigue. Simple type inference for structural polymorphism. In *Ninth International Workshop on Foundations of Object-Oriented Languages*, Portland, Oregon, 2002.

[4] Jacques Garrigue. Typing deep pattern-matching in presence of polymorphic variants. In *JSSST Workshop on Programming and Programming Languages*, Gamagori, Japan, March 2004.

[5] Jacques Garrigue, Benjamin Monate, Olivier Andrieu, Jun Furuse, Maxence Guesdon, and Stefano Zacchiroli. Lablgtk, an Objective Caml interface to Gtk+. Available at `http://wwwfun.kurims.kyoto-u.ac.jp/soft/olabl/lablgtk.html`.

[6] Koji Kagawa. Polymorphic variants in Haskell. In *Haskell '06 : Proceedings of the 2006 ACM SIGPLAN workshop on Haskell*, pages 37–47, New York, NY, USA, 2006. ACM Press.

[7] Didier Le Botlan and Didier Rémy. MLF : Raising ML to the power of System-F. In *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming*, pages 27–38, August 2003.

[8] Xavier Leroy, Damien Doligez, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. The Objective Caml system release 3.10, Documentation and user's manual, May 2007.

Types Simples, Logique et Coercions Implicites

Cody Roux[1]

1: Laboratoire Lorrain de Recherche en Informatique et ses Applications, LORIA - Campus Scientifique - BP 239 - 54506 Vandoeuvre-lès-Nancy Cedex, France
cody.roux@loria.fr

Résumé

Nous définissons un systeme général de coercions implicites dans les types simples et donnons un algorithme d'inférence de type. Ceci est réalisé grâce à une logique adéquate, inspirée de la logique linéaire, pour laquelle nous prouvons l'élimination des coupures.

1. Introduction

Les notions de coercion implicite et de sous-typage sont devenues importantes durant les dix dernières années, ceci à cause d'une motivation double. Les mathématiques telles qu'elles sont pratiquées font très souvent appel à la capacité de voir un même objet sous plusieurs points de vue différents ; il est probable que tout résultat profond utilise ce genre de changement de point de vue. Les preuves mathematiques contiennent évidamment aussi énormément d'information implicite. De façon parallèle, en informatique, il est intéressant que certains objets d'un type puissent être vus comme des objets d'un type différent, l'exemple fondateur étant celui de l'*héritage*, mais il se trouve également l'enjeu de l'utilisation modulaire de mêmes morceaux de code.

La bonne compréhension des coercions implicites est donc nécessaire pour continuer à développer l'objectif actuel de formalisation. Nous presentons un système général de coercions implicites dans les types simples et donnons un algorithme d'inférence des types en présence de coercions. Ceci est réalisé en appliquant l'isomorphisme de Curry-Howard et en presentant le sous-typage comme un theoreme d'une certaine logique. L'élimination des coupures dans cette logique correspond à la preuve de transitivité de la relation de sous-typage et nous permet de donner un algorithme d'inférence du sous-typage. Il est alors possible de donner un algorithme de typage en présence de coercions, qui dépend d'une notion de "type principal". Nous donnons alors des idées de développements futurs, notamment en ce qui concerne la cohérence du système.

Ce travail est issu du travail de Master de l'auteur. Nous voulons remercier M. Loic Pottier pour son exellent travail d'encadrement, et toute l'equipe Marelle, en particulier Ioana Pasca pour son support.

2. La notion de sous-typage

Dorénavant, nous nous plaçons dans le λ-calcul simplement typé (à la Church) défini de la manière suivante :

- L'ensemble des types :
$$\mathcal{T} := \mathcal{V} \mid \mathcal{T} \rightarrow \mathcal{T}$$
où $\mathcal{V}$ est un ensemble de variables de types
- L'ensemble des termes :
$$\Lambda := V \mid \lambda V : \mathcal{T}.\Lambda \mid (\Lambda\Lambda)$$
où V est un ensemble de variables de termes
- les règles de typage :

$$\frac{}{\Gamma, x\colon A \vdash x\colon A}$$

$$\frac{\Gamma \vdash f\colon A \to B \qquad \Gamma \vdash t\colon A}{\Gamma \vdash (f\ t)\colon B}$$

$$\frac{\Gamma, x\colon A \vdash t\colon B}{\Gamma \vdash \lambda x\colon A.t\colon A \to B}$$

Comme de coutume, nous noterons $A_1 \to A_2 \ldots A_{n-1} \to A_n$ pour $A_1 \to (A_2 \ldots (A_{n-1} \to A_n) \ldots)$ et $t\ u_1 \ldots u_n$ au lieu de $(\ldots (t\ u_1) \ldots) u_n$. Nous définissons de plus $t[x \leftarrow u]$ pour la substitution de la variable x par le terme u dans le terme t. Comme souvent dans de telles situations, nous adopterons la convention de Barendregt [2], qui permet de laisser toutes les questions d'α-conversion comme exercice au lecteur.

La définition du sous-typage peut s'exprimer comme une relation de préordre $\leq$ sur les types ainsi que la règle de typage suivante :

$$\frac{\Gamma \vdash t\colon A \qquad A \leq B}{\Gamma \vdash t\colon B}$$

Il est naturel d'imposer à cette relation d'être une relation de préordre. Cependant, cette relation peut très bien ne pas être antisymétrique ; on a alors $A \equiv B$ sans avoir pour autant $A = B$. Ceci ne pose en général pas de problème, sauf quand le sous-typage est défini en termes de coercions implicites, où il se pose la question de la *cohérence*. Nous en reparlerons dans la section 6.

La motivation informatique est déja claire dans ce qui précède : nous voulons être capable de manipuler un objet en fonction de ses capacités calculatoires. Si cet objet présente les mêmes aspects calculatoires qu'un objet d'un type différent, nous voulons être capables d'effectuer les mêmes opérations sur lui que nous aurions pu effectuer sur un objet du deuxième type. Un exemple phare de ce type de motivation est l'*héritage*.

La motivation mathématique est issue de la pratique quotidienne et informelle des mathématiques. Lorsque nous parlons d'un objet mathématique, il peut souvent être vu comme ayant deux aspects très différents. On peut par exemple voir un morphisme de structure comme une simple fonction (cet aspect rejoint alors la notion d'héritage mentionnée ci-dessus), ou un polynôme comme soit une liste (finie) de coefficients soit comme une fonction. Ce genre d'ambiguité est omniprésente en mathématique, et avoir la capacité de formaliser ce genre d'ambiguités semble être un enjeu important pour la capacité de formaliser de façon acceptable le corpus des mathématiques scolaires (un enjeu qui commence à devenir réaliste).

Il nous reste maintenant à trouver une définition de la relation de sous-typage et à donner l'algorithme de typage correspondant. nous allons pour cela utiliser le concept de *coercion implicite*. Malheureusement, nous verrons qu'il n'est pas facile de donner une définition intéressante et effective.

3. Des Coercions dans les Types Simples

Les types dépendants étant trop complexes pour avoir un algorithme simple de typage prenant en compte les coercions implicites nous nous intéressons au λ-calcul simplement typé. Ce genre de problème est abordé dans [14] avec une restriction sur les jugements de sous-typages possibles : les contraintes de sous-typage ne peuvent se faire entre deux types construits, nous ne pouvons par exemple avoir $A \to B \leq C \to D$ sans avoir $C \leq A$ et $B \leq D$, nous pensons qu'il est intéressant de lever cette restriction, notamment pour les exemples issus de motivation mathematiques. Une présentation des algorithmes utilisés dans cette situation peut se trouver dans [15].

Nous avons décidé d'adopter un point de vue proche de celui de [11] et de faire intervenir la notion d'*isomorphisme de Curry-Howard*. Rappelons en quoi consiste ce concept pour le λ-calcul simplement typé : l'isomorphisme de Curry-Howard construit une bijection entre les propositions de la logique implicative minimale et les types d'une part, et entre les preuves de ces propositions et les termes des types correspondants d'autre part.

Rappelons les règles de la logique minimale :

$$\frac{}{\Gamma, A \vdash A}\ \mathbf{Ax}$$

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A{\to}B}\ \mathbf{Abs}$$

$$\frac{\Gamma \vdash A{\to}B \qquad \Gamma \vdash A}{\Gamma \vdash B}\ \mathbf{Cut}$$

Sous cet isomorphisme, les coercions définies par l'utilisateur sont vues comme des *hypothèses* qui constituent un contexte $\mathcal{C}$ et construire une coercion du terme t de type A vers un terme de type B revient alors à donner une démonstration de

$$\mathcal{C}, A \vdash B$$

dans une logique à définir, mais dont les règles doivent être admissibles en logique minimale implicative. Il sera alors possible, grâce à l'isomorphisme, de reconstruire un terme t' de type B à partir du terme t, c'est ce terme qui sera le coercé de A vers B. De quoi doit avoir l'air cette logique ? Certaines conditions s'imposent.

1. L'hypothèse A doit forcément être "consommée" une et une seule fois par la preuve. Il est en effet impensable qu'une coercion d'un terme t de type A vers un terme t' de type B ne fasse pas intervenir t. En particulier, si une des coercions f est de type $U{\to}V$ et $B \equiv U{\to}V$ alors le terme t' ne doit pas être constitué du seul terme f.
2. L'ordre des arguments ne doit pas être dérangé : si par exemple $A \equiv U{\to}U{\to}V$, $B \equiv U{\to}U{\to}V'$ et qu'il y a dans $\mathcal{C}$ une coercion f de type $V{\to}V'$, le terme t' formé ne doit pas être $\lambda x\colon U.\lambda y\colon U.f\ (t\ y\ x)$, mais $\lambda x\colon U.\lambda y\colon U.f\ (t\ x\ y)$.
3. Il ne doit pas non plus être possible de "dupliquer" l'hypothèse A. Nous ne voulons pas, par exemple, former de coercion entre les types $A{\to}A{\to}B$ et $A{\to}B$ sans avoir défini de coercion appropriée.

Il est intéressant de noter que ces restrictions sont sujettes a discussion. Nous savons, par exemple, qu'un étudiant confronté à une notation fonctionnelle dont les arguments ont étés inversés peut nénmoins comprendre le sens de l'expression dans le cas où l'expression vue n'aurait pas de sens. Il est possible qu'un relachement des conditions ci-dessus permettent de simuler ce genre de comportement et donc de simplifier l'interaction d'un utilisateur avec un prouveur de théorèmes par exemple.

Il est également naturel de demander à la partie $\mathcal{C}$ du contexte de ne contenir que des coercions (définies par l'utilisateur par exemple), c'est à dire des types fonctionnels. Cependant, il est possible d'imaginer un contexte qui contient des types non fonctionnels, qui pourraient représenter des arguments à appliquer de façon implicite dès que possible, par exemple l'univers dans lequel on se place. Nous aurions alors, si U représente le type de cet univers, une coercion entre toutes les fonctions $U{\to}A$ et A. Encore une fois, ce genre de convention est adoptée de façon systematique en mathématiques. Nous n'explorons pas plus en avant ce point de vue dans la suite du papier, et dorénavent, on suppose que $\mathcal{C}$ ne contient que des arguments fonctionnels.

3.1. Les règles de la logique des coercions

Certaines de ces conditions sont remniscientes de la *logique linéaire* [9] et de la *relevance logic*[1]. Cependant ces logiques sont encore un peu trop expressives. Après plusieurs tentatives, nous avons choisi de définir le système qui suit.

Nous séparons le contexte en deux parties. Une partie contient les coercions, l'autre contient une unique hypothèse, qui correspond au type du terme dont on veut donner une coercion. On définit :

Définition 1 *Une* proposition *est un terme sur la signature* $\mathcal{P} := \mathcal{V} \mid \mathcal{P}{\to}\mathcal{P}$ *Avec* $\mathcal{V}$ *un ensemble de variables.*

Définition 2 *Un* contexte *est un couple* $\langle \mathcal{C}, H \rangle$ *avec* $\mathcal{C}$ *un ensemble de propositions de la forme* $P_1{\to}P_2$ *et* P_1, P_2, H *des propositions.*

On notera $\mathcal{C}; H$ au lieu de $\langle \mathcal{C}, H \rangle$ et $\mathcal{C}, T$ au lieu de $\mathcal{C} \cup \{T\}$.

On définit ensuite la *logique des coercions* $\vdash_{\mathbf{coer}}$ avec les règles suivantes :

$$\frac{}{\mathcal{C}; H \vdash_{\mathbf{coer}} H}\ \mathbf{Ax}$$

$$\frac{\mathcal{C}, A{\to}B; H \vdash_{\mathbf{coer}} A}{\mathcal{C}, A{\to}B; H \vdash_{\mathbf{coer}} B}\ \mathbf{App}$$

$$\frac{\mathcal{C}; C \vdash_{\mathbf{coer}} A \qquad \mathcal{C}; B \vdash_{\mathbf{coer}} D}{\mathcal{C}; A{\to}B \vdash_{\mathbf{coer}} C{\to}D}\ \mathbf{Contr}$$

$$\frac{\mathcal{C}; A \vdash_{\mathbf{coer}} B \qquad \mathcal{C}; B \vdash_{\mathbf{coer}} C}{\mathcal{C}; A \vdash_{\mathbf{coer}} C}\ \mathbf{Cut}$$

Nous écrirons $\vdash$ au lieu de $\vdash_{\mathbf{coer}}$ lorsque le contexte sera clair.

La première règle permet de former la coercion identité. On remarque qu'elle ne s'applique que à la partie droite du contexte : c'est ceci qui permet de remplir la condition numéro 1 détaillée dans la section précédente. La seconde règle est la règle fondamentale, c'est elle qui permet d'appliquer les coercions. La troisième règle permet d'obtenir la propriété de contravariance, sa forme nous donne les garanties de la seconde et la troisième condition.

Enfin la coupure garantit la transitivité de la relation. Il n'est pas évident que cette règle soit nécessaire ; en effet, beaucoup de logiques admettent un théorème d'elimination des coupures, c'est à dire que la règle **Cut** est admissible dans cette logique. Ici cependant la règle n'est pas admissible ; ceci vient d'un manque de symétrie dans les règles, il n'est pas possible de les appliquer à gauche du $\vdash$ comme dans un calcul des séquents. Il est possible de compléter les trois autres règles afin d'avoir un véritable calcul des séquents et de prouver l'élimination des coupures, ce sera l'objet de la section 4. Ceci est nécessaire pour prouver la décidablilité de la déduction dans cette logique.

Nous pouvons faire la remarque suivante : la notation $\mathcal{C}; H$ n'est pas sans rappeler les systèmes à *bénitier* (voir par exemple [10]) dans lesquels une partie du séquent est mise à l'écart du reste des hypothèses. Le but de ces systèmes est de mieux gérer la recherche de preuve. Cependant nous n'avons ici aucune régle qui permet de "remplir" le bénitier, et nos motivations sont différentes. Nous reviendrons sur cette analogie plus tard.

Définition 3 *Nous dirons qu'un séquent est* dérivable *si il existe une preuve de ce séquent dans la logique des coercions.*

3.2. les termes

Examinons la forme des termes produits par ces règles sous les lumières de l'isomorphisme de Curry-Howard : les règles correspondantes avec les annotations de λ-termes sont

$$\frac{}{\mathcal{C}; t\colon H \vdash t\colon H}\ \mathbf{Ax}$$

$$\frac{\mathcal{C}, f\colon A{\to}B; t\colon H \vdash u\colon A}{\mathcal{C}, f\colon A{\to}B; t\colon H \vdash (fu)\colon B}\ \mathbf{App}$$

$$\frac{\mathcal{C}; x\colon C \vdash a\colon A \qquad \mathcal{C}; y\colon B \vdash d\colon D}{\mathcal{C}; t\colon A{\to}B \vdash \lambda x\colon C.d[y{\leftarrow}(ta)]\colon C{\to}D}\ \mathbf{Contr}$$

$$\frac{\mathcal{C}; x\colon A \vdash b\colon B \qquad \mathcal{C}; y\colon B \vdash c\colon C}{\mathcal{C}; x\colon A \vdash c[y{\leftarrow}b]\colon C}\ \mathbf{Cut}$$

Donnons un exemple. Soit A une variable de type et $\mathcal{C}$ l'ensemble de coercions $\{A{\to}(A{\to}A)\}$. Nous voulons montrer $\mathcal{C}; A \vdash A{\to}A{\to}A$. Nous pouvons effectuer la dérivation suivante :

$$\dfrac{\dfrac{\dfrac{}{\mathcal{C}; A \vdash A}\ \mathbf{Ax}}{\mathcal{C}; A \vdash A{\to}A}\ \mathbf{App} \qquad \dfrac{\dfrac{}{\mathcal{C}; A \vdash A}\ \mathbf{Ax} \qquad \dfrac{}{\mathcal{C}; A \vdash A}\ \mathbf{Ax}}{\mathcal{C}; A{\to}A \vdash A{\to}A{\to}A}\ \mathbf{App}}{\mathcal{C}; A \vdash A{\to}A{\to}A}\ \mathbf{Contr}\ \mathbf{Cut}$$

En rajoutant les annotations, on obtient le terme suivant :

$$\phi\colon A{\to}(A{\to}A); t\colon A \vdash \lambda x\colon A.\lambda y\colon A.\phi[(\phi t)x]y\colon A{\to}A{\to}A$$

Cela correspond au terme que l'on désire avoir lorsqu'une coercion est éffectuée entre A et $A{\to}A{\to}A$. On remarque que, si on efface les annotations de type et la fonction ϕ, on obtient le terme (non typé) $\lambda x.\lambda y.txy \twoheadrightarrow_\eta t$. Il semble naturel que, une fois effacée l'information de typage ainsi que les coercions employées, le terme obtenu soit le terme de départ, modulo η-équivalence. Nous verrons qu'il en est ainsi (théorème 1) de toutes les coercions construites dans notre système.

Dans toute la suite, soit $\mathcal{C}$ un ensemble de coercions (variables de type $U{\to}V$), t une variable de type A, et u un terme de type B.

Lemme 1 *Si dans la logique des coercions $\mathcal{C}; t\colon A \vdash u\colon B$ est dérivable alors toute occurrence d'une variable de coercion $f \in \mathcal{C}$ dans u est en partie gauche d'une application. De plus u est de la forme $\lambda x\colon T.u'$ ou $(f\ u')$ avec $f \in \mathcal{C}$.*

Démonstration
Induction évidente sur la dérivation de u. □

Lemme 2 *Avec les notations précédentes, si $\mathcal{C}; t\colon A \vdash u\colon B$ est dérivable alors l'unique occurrence de t dans u est soit à gauche soit à droite d'une application, soit $u = t$.*

Démonstration
Encore une induction immédiate sur la dérivation de u. □

Définition 4 *On définit* l'effacement $\mathit{eff}\colon \Lambda{\to}\Lambda^{\mathbf{pur}}$ *inductivement sur la structure d'un terme de la façon suivante :*

- $\mathit{eff}(x) = x$ *si x est une variable*
- $\mathit{eff}(\lambda x\colon A.t) = \lambda x.\mathit{eff}(t)$
- $\mathit{eff}(ft) = \mathit{eff}(t)$ *si $f \in \mathcal{C}$, $(\mathit{eff}(f)\mathit{eff}(t))$ sinon.*

Nous pouvons alors énoncer le théorème évoqué ci-dessus :
Proposition 1 *Supposons que $\mathcal{C}; t : A \vdash u : B$. Alors* l'effacement $\mathit{eff}(u)$ *est η-réductible à la variable t.*
Tout terme de cette forme sera dit sous *forme coercive.*

Démonstation : Par induction sur la dérivation de $\mathcal{C}; t : A \vdash u : B$.

- Si la dernière règle utilisée est **Ax** alors $t = u$ et donc $\mathit{eff}(u) \twoheadrightarrow_\eta t$.
- Si la dernière règle utilisée est **App** alors $u = (fu')$ et on a $\mathit{eff}(u) = \mathit{eff}(u') \twoheadrightarrow_\eta t$.
- Si la dernière règle est **Contr** alors $u = \lambda x : T.u'(tu''(x))$ et donc par hypothèse d'induction $\mathit{eff}(u) \twoheadrightarrow_\eta \lambda x.tx{\to_\eta}t$.
- Si la dernière règle est **Cut** alors le résultat est clair.

□

On a même une réciproque :

Proposition 2 *Soit u un λ-terme typé avec une variable libre $t\colon A$ tel que :*

1. *le terme u est de type B dans le contexte $\mathcal{C}, t\colon A$*
2. *u est de forme coercive, c'est à dire $\mathit{eff}(u) \twoheadrightarrow_\eta t$*

alors $\mathcal{C}; t\colon A \vdash u\colon B$ est dérivable.

Démonstration : Nous raisonnons par induction sur la structure du terme u. Le terme u est soit t, soit une abstraction, soit une application.

- Si $u = u_1u_2$ est une application, alors $\mathit{eff}(u) \twoheadrightarrow_\eta t$ implique $u_1 = f$ pour un certain $f \in \mathcal{C}$. Par hypothèse d'induction, on peut prouver le jugement de typage $\mathcal{C}; t\colon A \vdash u_2\colon B'$ et donc $\mathcal{C}; t\colon A \vdash fu\colon B$ par application de la règle **App**.
- Si $u = \lambda x\colon T.u'$ alors on a $\mathit{eff}(u') \twoheadrightarrow_\eta tx$ et donc u' est de la forme $u_1(u_2(t)u_3(x))$ avec u_1, u_2, u_3 sous forme coercive. On peut alors appliquer l'hypothèse d'induction pour construire $\lambda x\colon T.u_1(t\ u_3(x))$ grâce à **Contr**, puis utiliser **Cut** pour construire u.

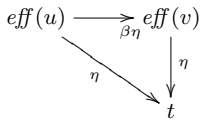
□

Ceci nous permet de comprendre la structure des coercions que produit notre système logique. Cette structure est stable par réduction β et η, et les termes produits sont fortement normalisants :

Lemme 3 *Si avec les notations précédentes u est sous forme coercive alors*

1. *u est fortement normalisant*
2. *pour tout λ-terme v, $u \twoheadrightarrow_{\beta\eta} v \Rightarrow$ v est sous forme coercive.*

Démonstration : Le terme u étant typable dans le λ-calcul simplement typé avec le contexte $\mathcal{C}, t\colon A$, il est fortement normalisant d'après le théorème de normalisation forte (voir [3]). D'autre part, supposons que l'on ait $\mathit{eff}(u) \twoheadrightarrow_\eta t$ avec t une variable. Alors, grâce à la propriété de Church-Rosser de la $\beta\eta$-réduction du λ-calcul [2], on a pour tout terme v $\beta\eta$-équivalent à u :



Ce qui montre la seconde partie du lemme.

Remarquons que toute la puissance de la normalisation forte du λ-calcul simplement typé n'est pas neccesaire : les termes n'ayant ici qu'une seule occurence pour chaque variable, la normalisation est bien plus simple à montrer.

4. L'Elimination des Coupures

Il reste maintenant à donner une procédure de décision, ce qui permettra, étant donné $\mathcal{C}, A, B$ de voir si il existe une coercion de A vers B. Pour cela, il est nécessaire d'éliminer la règle **Cut**, car celle-ci peut s'appliquer à chaque étape d'une preuve et rend donc difficile la recherche d'un algorithme de décision. Nous l'avons précisé, il faut pour cela rajouter des règles à notre logique pour donner un symétrique à droite de chaque règle qui s'applique à gauche, c'est à dire **App** et **Contr**, la règle **Ax** étant déja symétrique. Ceci est analogue à l'utilisation des symmetries pour montrer l'élimination des coupures dans le calcul des sequents de Genzen (mais de façon bien plus simple, puisque nous utilisons déja un théorème de normalisation forte).

On rajoute donc les règles suivantes (annotées) à **Coer** :

$$\frac{\mathcal{C}, f\colon A{\to}B; t\colon B \vdash c\colon C}{\mathcal{C}, f\colon A{\to}B; u\colon A \vdash c[t{\leftarrow}(fu)]\colon C}\ \textbf{Appin}$$

Avec u une variable *fraîche* c'est à dire non liée dans c.

$$\frac{\mathcal{C}'; x\colon A \vdash d\colon D \qquad \mathcal{C}'; y\colon E \vdash b\colon B \qquad \mathcal{C}'; z\colon C \vdash h\colon H}{\mathcal{C}'; t\colon D{\to}E \vdash h\,[z{\leftarrow}(f\ \lambda x\colon A.b[y{\leftarrow}(td)])]\,:\,H}\ \textbf{Contrin}$$

Avec $\mathcal{C}' = \mathcal{C}, f\colon (A{\to}B){\to}C$.

Nous pouvons vérifier que ces règles sont admissibles dans la logique des coercions avec la coupure.

On veut alors montrer :

Théorème 1 *Soit un séquent $\mathcal{C}; t\colon A \vdash u\colon B$ dérivable. Il existe une dérivation du séquent $\mathcal{C}; t\colon A \vdash u'\colon B$ avec u' la forme normale de u qui n'utilise pas la règle **Cut**.*

Nous allons utiliser notre connaissance de la forme du terme u pour prouver ce théorème. Comme notre calcul défini est un sous-calcul du lambda calcul simplement typé, le théorème de "subject reduction" du lambda calcul simplement typé s'applique et nous assure que tout terme coercif, si il est mis sous forme normale, garde le même type que celui de départ. Etant donné une coercion u, on peut la normaliser d'après le lemme 3, puis construire le terme normal dans la logique des coercions en utilisant la proposition 2. Il suffit donc de démontrer le théorème ci-dessus pour les termes u en forme β-normale.

Lemme 4 *Supposons, avec les notations précédentes, que $u = c(t)$ et $c(t) = c_0(d_0(t))$, avec c_0 et d_0 sous forme coercive. Alors si on peut construire $c_0(x)$ et $d_0(t)$ sans la règle **Cut**, et que $c(t)$ est en forme normale, on peut construire le terme $c(t)$ sans la règle **Cut**.*

Démonstration : Nous procédons par induction sur la taille du terme d_0. Si $d_0(t) = t$ alors $c(t) = c_0(t)$ peut être construit sans coupure par hypothèse. Sinon d'après la forme des règles d'inférence de la logique des coercions, $d_0(t)$ peut être de deux formes possibles :

- $d_0(t) = (fd_1(t))$ avec $f \in \mathcal{C}$ d'après le lemme 1. On peut alors appliquer la règle **appin** au terme $c_0(t)$ pour obtenir le terme $c_0(ft)$ sans coupure. On peut alors appliquer l'hypothèse d'induction appliquée à d_1 et $c_0(ft)$ pour obtenir le terme $c(t) = c_0(fd_1(t))$
- $d_0(t) = \lambda x : T.d_1(d_2(t)d_3(x))$. Nous pouvons en effet observer que la seule façon de construire un terme de la forme $\lambda x\colon T.u$ est par application de la règle **Contr** puis d'application des seules règles **Appin** et **Contrin**. Après application de la règle **Contr** le terme est égal à $\lambda x : T.d_1(t\ d_3(x))$, puis les règles **Appin** et **Contrin** n'effectuent que des remplacements sur la variable t.
  Alors d'après le lemme 2, soit $c_0(t) = t$ et on peut conclure, soit l'occurrence de t dans $c_0(t)$ est en partie droite d'une application, soit elle est en partie gauche. Elle ne peut en fait pas être en partie gauche d'une application, car alors $c_0(d_0(t))$ contiendrait un redex enjendré par l'abstraction dans d_0. Cette occurrence est donc en partie droite d'une application, et d'après le lemme 1, on peut écrire $c_0(t) = c_1(ft)$ pour un certain $f \in \mathcal{C}$. Il est donc possible d'appliquer la règle **contrin** pour construire le terme

$$c_2(t) = c_1(f\ \lambda x : T.d_1(t\ d_3(x)))$$

  On peut alors conclure grâce à l'hypothèse d'induction appliquée à d_2 et c_2.

Ceci nous permet de démontrer le théorème 1 : par induction sur la taille du terme en utilisant le lemme 4. □
A partir de maintenant, nous noterons $A \vdash B$ au lieu de $\mathcal{C}; A \vdash B$ si l'ensemble de coercions n'est pas ambigu.

5. L'Algorithme d'Inférence de Type

Nous pouvons maintenant remarquer la chose suivante : en regardant les règle de déduction du bas vers le haut (*bottom-up*) la taille des types diminue pour les règles **Contr** et **Contrin**, et elle est bornée par la taille des types dans $\mathcal{C}$ pour les règles **App** et **Appin**. Seule la règle **Cut** posait problème, il suffit d'appliquer le théorème 1 pour s'en débarasser. Lorsque nous voulons démontrer un séquent de la forme $\mathcal{C}; A \vdash B$ en appliquant les règles ci-dessus, les types apparaissant dans les séquents seront de taille bornée. Nous nous déplaçons donc dans un ensemble fini de séquents possibles, ceci nous donne un moyen de construire une procédure de décision pour cette logique.

5.1. La décidabilité de *coer*

Etant donné un ensemble de coercions $\mathcal{C}$ on peut donner l'algorithme suivant pour décider, étant donné deux types A et B si $\mathcal{C}; A \vdash B$ et, le cas échéant, donner le terme de coercion associé :

$\textsc{Colog}(A\colon \textsc{type}, B\colon \textsc{type}, L\colon \textsc{seqlist}, t\colon \textsc{var})\colon \textsc{term}$
  if $\langle A, B\rangle \in L$
    then return FAIL
    else $L \leftarrow \langle A, B\rangle :: L$
      try
      Ax$(A = B)$
        return t
      App$(f\colon C \to B \in \mathcal{C})$
        return $f(\textsc{Colog}(A, C, L, t))$
      Contr$(A = A_1 \to A_2, B = B_1 \to B_2)$
        return let $y = (t\ \textsc{Colog}(B_1, A_1, L, x))$ **in** $\lambda x\colon B_1.\textsc{Colog}(A_2, B_2, L, y)$
      Appin$(g\colon A \to C \in \mathcal{C})$
        return let $y = (g\ t)$ **in** $\textsc{Colog}(C, B, L, y)$
      Contrin$(A = A_1 \to A_2, f : (B_1 \to B_2) \to C \in \mathcal{C})$
        return
          let $z =$
          $(\textbf{let}\ y = (t\ \textsc{Colog}(B_1, A_1, L, x))\ \textbf{in}\ (f\ \lambda x\colon B_1.\textsc{Colog}(A_2, B_2, L, y)))$
          in $\textsc{Colog}(C, B, L, z)$

On décide alors si $\mathcal{C}; A \vdash B$ en appelant `Colog`$(A, B, [], t)$, si c'est le cas alors l'algorithme renvoie un terme de preuve, sinon il renvoie `FAIL`.

Le **try** dans cette procédure n'est pas déterministe : il faut essayer toutes les possibilités afin de construire un *arbre d'appels récursifs*. Une branche de cet arbre est alors *élagué* si un appel se termine en échec (c'est le *fail*). On veut alors récuperer les noeuds non coupés de cet arbre. Cette méthode de programmation peut rappeler les méthodes déclaratives présentes dans le langage `PROLOG` par exemple.

Il n'est pas évident que cet algorithme termine. Cependant, la terminaison est garantie par le fait que la taille des types d'entrée dans les appels de la procédure sont bornées, et que la liste passée en argument grandit strictement à chaque appel. Il arrive donc un endroit dans chaque branche dans lequel soit le séquent $\langle A, B\rangle$ est présent dans la liste, soit $A = B$. L'arbre des appels est donc à chemins finis et chaque arrête à un nombre de fils fini, l'arbre est donc fini, l'algorithme termine.

5.2. L'Algorithme de Typage

Supposons donné un λ-terme avec annotations de types et un ensemble de coercions. Nous voulons donner un algorithme qui détermine si le terme est typable dans notre système de types avec coercions et si c'est le cas, donner son type.

Remarquons d'abord que notre système est suffisamment puissant pour modéliser la surcharge de fonctions. Par exemple, considérons deux fonctions, $+_{\mathbf{R}} : \mathbf{R}\to\mathbf{R}\to\mathbf{R}$ et $+_{\mathbf{N}} : \mathbf{N}\to\mathbf{N}\to\mathbf{N}$ que nous voulons dénoter avec un seul symbole de fonction, $+$. Il suffit alors de créer un type "intersection" I_+ habité par le seul terme $+$ ainsi que deux coercions $c_1\colon I_+\to\mathbf{R}\to\mathbf{R}\to\mathbf{R}$ et $c_2\colon I_+\to\mathbf{N}\to\mathbf{N}\to\mathbf{N}$ qui à $+$ associent $+_{\mathbf{R}}$ et $+_{\mathbf{N}}$ respectivement. I_+ étant un type atomique n'apparaissant nulle part d'autre que dans le type du terme $+$ il est clair que dans tout terme bien typé contenant $+$ appliqué à des arguments une des deux coercions aura étée utilisée. Notons que I_+ peut être vu comme le type $\mathbf{N}\to\mathbf{N}\to\mathbf{N} \wedge \mathbf{R}\to\mathbf{R}\to\mathbf{R}$, ce qui permet de donner une idée de l'intérêt d'étendre la logique des coercions avec des connecteurs $\wedge$, $\vee$ etc, ou d'utiliser une extension du λ-calcul du type de celui étudié dans [5].

Pour typer un terme du λ-calcul il est nécessaire de procéder inductivement sur la structure du terme. Dans le cas sans coercions (et sans inférence de types à la curry) l'algorithme est simple : dans le cas d'une abstraction $\lambda x\colon A.t$, il suffit de typer le terme t avec le contexte $x\colon A$ et de renvoyer le type $A\to B$ si t est de type B. Dans le cas d'une application $(t\ u)$, il faut typer t et u. Le terme est bien typé dans le cas où le type de t est de la forme $A\to B$ et le type de u est A. Le type du terme est alors B.

Les choses sont évidemment plus compliquées en présence de coercions, un terme pouvant avoir éventuellement une infinité de types possibles. Par exemple dans le cas d'une unique coercion $f\colon A\to(A\to A)$, tout terme t auquel on peut attribuer le type A peut également être attribué le type $A\to A$, $A\to A\to A$, $A\to A\to A\to A$, etc.

Il n'est donc *a-priori* pas possible d'énumérer tous les types possibles des sous-termes d'un terme afin de pouvoir choisir les types permettant de typer le terme complet. Il faut cependant chercher à effectuer cette énumération, car en effet il peut arriver que la première coercion trouvée ne soit pas la bonne pour pouvoir typer le terme complet. Par exemple, si on considère l'ensemble de coercions $\{c_1\colon H\to A\to B, c_2\colon H\to A\to A\to B\}$ et le terme $f\ a_1\ a_2$ avec $f\colon H$ et $a_1, a_2\colon A$ alors on peut typer le sous terme $f\ a_1$ grâce à la coercion c_1 pour lui donner le type B. Il est alors impossible de typer le terme complet sans revenir en arrière pour donner un type différent au terme $f\ a_1$, (faire du *backtracking*).

Une solution naïve serait de chercher à typer une série d'applications en une étape, c'est à dire (pour cet exemple) chercher à typer a_1 et a_2 (ici seul le type A est possible) et ensuite chercher a trouver un type pour f de la forme $A\to A\to X$ pour X un certain type. Cependant ceci pose encore problème, comme le montre l'exemple suivant :

On se donne les coercions $c_1\colon I_+\to(N\to N\to N), c_2\colon I_+\to(D\to D\to R), c_3\colon N\to D$ et on se donne les variables $+\colon I_+, \times\colon R\to R\to R, n\colon N, m\colon N, r\colon R$. Nous voulons typer le terme $\times\ (+\ n\ m)\ r$. Pour typer ce terme avec l'idée ci-dessus, nous cherchons à typer $+\ n\ m$, ce qui est possible et donne le type N puis nous cherchons une coercion de $R\to R\to R$ vers $N\to R\to X$ pour un certain X. Cette coercion n'existe pas. Il faut donc un algorithme qui permette de revenir en arrière afin d'essayer tous les types possibles pour les arguments des fonctions.

Il existe en général une infinité de types possibles dans ce cas. Cependant, il suffit de considérer un nombre fini de types grâce au lemme suivant :

Lemme 5 *Soit $\mathcal{C}$ un ensemble de coercions et F un type. Soit I un ensemble d'indices ($I\subseteq\mathbb{N}$) et $(F_1^i)_{i\in I}$ et $(F_2^i)_{i\in I}$ l'ensemble des types tels que $F\vdash_{\mathbf{coer}} F_1^i\to F_2^i$.*

Il existe $J\subset I$ fini tel que $\forall i\in I, \exists j\in J$ tel que

$$F_1^i\vdash_{\mathbf{coer}} F_1^j$$

et

$$F_2^j\vdash_{\mathbf{coer}} F_2^i$$

Nous pouvons même donner explicitement la famille $(F_\epsilon^j)_{j\in J,\epsilon=1,2}$: On peut évidemment supposer qu'il existe un entier naturel n tel que $J=\{k\in\mathbb{N}\mid k\leq n\}$. Si $F\equiv A\to B$ alors $F_1^1=A$ et $F_2^1=B$; pour tout $j>1$ les F_1^j sont les types V et les F_2^j sont les types W tels que $\exists f\in\mathcal{C}$ avec $f\colon U\to(V\to W)$; si F est atomique, alors les F_1^j et F_2^j sont de la deuxiemme forme pour tout $j\in J$.

Les F_ϵ^j ne dépendent donc que de $\mathcal{C}$, à part éventuellement F_1^1 et F_2^1 qui peuvent dépendre de F.

Ces types correspondent à une certaine notion de "type principal" c'est à dire qu'il suffit de connaitre ces types pour effectuer le typage.

Démonstration : Nous démontrons le lemme par induction sur la longeur de la dérivation de $F\vdash F_1^i\to F_2^i$ dans le système avec les trois règles **Ax**, **App**, **Contr** et la coupure :

- Cas **Ax** : $F_1^1\to F_2^1=F=F_1^i\to F_2^i$, donc $F_1^i\vdash F_1^1$ et $F_2^1\vdash F_2^i$ de façon triviale.
- Cas **App** : $\exists f\in\mathcal{C}$ tel que $f\colon H\to F_1^i\to F_2^i$, donc $i\in J$ et on peut conclure.
- Cas **Contr** : $F=F_1^1\to F_2^1$ et on a $F_1^i\vdash F_1^1$ et $F_2^1\vdash F_2^i$.
- Cas **Cut** : On a $F\vdash H$ et $H\vdash F_1^i\to F_2^i$. Par hypothèse d'induction appliquée à $H\vdash F_1^i\to F_2^i$, on a soit $H\equiv H_1\to H_2$ et $F_1^i\vdash H_1$, soit $H\equiv H_1\to H_2$ et $\exists j\in J, j>1$ tel que $F_1^i\vdash H_1^j$, soit H atomique et $\exists j\in J$ tel que $F_1^i\vdash H_1^j$. Dans les deux derniers cas, il existe $j'\in J$ ($j'=j$ ou $j+1$) tel que $F_1^{j'}=H_1^j$.

  Il suffit donc de traiter le premier cas. Dans ce cas, par hypothèse d'induction appliquée à $F\vdash H_1\to H_2$, il existe $k\in J$ tel que $H_1\vdash F_1^k$ et donc $F_1^i\vdash F_1^k$ par transitivité du sous-typage. On vérifie sans difficultés que $F_2^{j'}\vdash F_2^i$.

L'algorithme proposé est le suivant : on se donne un contexte avec une liste `VAR` de variables x indexées par leur type T_x. Etant donné un type T, les éléments des familles finies $(T_1^j)_{j\in J}$

et $(T_2^j)_{j\in J}$ définies dans le lemme précédent seront notées, pour $j \in J$, $\mathtt{Ar}_1^j(T)$ et $\mathtt{Ar}_2^j(T)$ respectivement.

COTYPE(t: TERM, VAR)
  match t **with**
    $x \in$ VAR **return** T_x
    $\lambda x\colon T.t'$ **return** $T \to$ COTYPE$(t', \langle x, T\rangle :: \text{VAR})$
    $(t_1\ t_2)$ **try**
    local $=$ COLOG(COTYPE$(t_2, \text{VAR}), \mathtt{Ar}_1^j(\text{COTYPE}(t_1, \text{VAR})), [], t)$ **with** $j \in J$
    if *local* $\neq$ FAIL **return** $\mathtt{Ar}_2^j(\text{COTYPE}(t_1, \text{VAR}))$
    else return FAIL

Le **try** ici est encore non déterministe : il essaye toutes les possibilités pour $j \in J$. C'est en ceci que l'algorithme effectue du *backtracking*. La correction de cet algorithme est facile à vérifier. Il faut vérifier que cet algorithme est *complet* c'est à dire qu'un terme est typable en présence de coercions seulement si l'algorithme renvoie un résultat (différent de `FAIL`). La complétude de cet algorithme est garantie par le lemme 5, pour typer $(t_1\ t_2)$, il suffit de vérifier que le type de l'argument se coerce vers un des F_1^j pour déterminer si l'application est typable, pour tous les types possibles de t_1 ; en effet si il existe une coercion du type de t_1 vers un type de la forme $T_{t_2} \to X$ avec T_{t_2} un type possible de t_2. Par le lemme 5, si un tel type existe, alors $T_{t_2} \vdash F_1^j$ pour un certain $j \in J$. L'ensemble J étant toujours fini, nous pouvons énumérer tous les cas.

6. La cohérence

Lorsqu'on introduit un système de coercions, il est nécessaire de garantir que toute coercion entre deux types A et B est unique, c'est à dire

$$\forall c_1, c_2\colon A \to B \text{ coercions},\ c_1 =_{\beta\eta} c_2$$

Ceci est clairement faux dans notre système, en effet il est possible de définir $\mathcal{C} = \{f\colon A \to B, g\colon A \to B\}$ ou encore $\mathcal{C} = \{f\colon A \to A\}$ dans lequel cette unicité n'est pas présente. Il est cependant intéressant de se demander quelles limitations sur les éléments de $\mathcal{C}$ permettent d'avoir cette propriété.

Définition 5 *Soit $\mathcal{C}$ un ensemble de coercions. On dit que $\mathcal{C}$ est* cohérent *si pour tout types U, V et toute dérivation $t\colon U \vdash u_1\colon V$ et $t\colon U \vdash u_2\colon V$ dans la logique des coercions, alors $u_1 =_{\beta\eta} u_2$.*

Nous conjecturons ce résultat initial :

Conjecture 1 *Supposons que $\mathcal{C} = \{f\colon A \to B\}$ et que $A \not\equiv B$. Alors $\mathcal{C}$ est cohérent.*

La démonstration de cette proposition nous échappe pour l'instant, elle semble cependant accessible ; dans le cas où A n'est pas un sous terme de B qui n'est lui-même pas un sous terme de A, le résultat découle du résultat de cohérence dans [11].

Une conjecture un peu plus ambitieuse dont la vérité est moins certaine peut se formuler ainsi : On considère un ensemble $\mathcal{C}$ de coercions, et on remplace les coercions de la forme $f\colon (A \to B) \to (C \to D)$ par les coercions $f_1\colon C \to A$ et $f_2\colon B \to D$ ainsi de suite jusqu'à qu'il n'y ait plus que des coercions de type $A \to B$ ou $A \to (B \to C)$ ou $(A \to B) \to C$ avec A, B, C atomiques. On note $\mathcal{C}^*$ l'ensemble de coercions ainsi obtenu et on regarde $\mathcal{C}^*$ comme un graphe dans lequel les sommets sont les sources et buts des coercions et les arrêtes sont les coercions. On formule alors la conjecture :

Conjecture 2 *$\mathcal{C}$ est cohérent si et seulement si $\mathcal{C}^*$ est* sans cycles.

La seconde conjecture implique trivialement la première, et il est clair que $\mathcal{C}^*$ sans cycles est une condition neccesaire (tout cycle dans $\mathcal{C}^*$ permet de construire deux coercions distinctes). Il suffirait donc de montrer qu'elle est suffisante. Certains résultats concernant la cohérence ont étés démontrés dans [4] et [6], ils se généralisent peut-être à la situation présente.

Il est également remarquable que les systèmes à bénitier évoqués plus haut ont étés cr'e'es dans le but d'identifier certaines preuves $\beta\eta$-équivalentes. Certains résultats dans ce domaine pourraient donc peut-être s'appliquer dans cette situation.

Il peut aussi être intéressant d'étudier la situation lorsqu'il existe des équations entre coercions, par exemple $f \circ g = id$ avec $f, g \in \mathcal{C}$ et id l'identité, ou d'essayer de voir quelles équations il faut rajouter à un ensemble de coercions afin de le rendre cohérent. Ces questions sont la continuation logique du présent travail.

7. Conclusion

Nous avons détaillé un système de sous-typage dans le cas du λ-calcul simplement typé, et montré la décidabilité de la relation de sous-typage, et donné un algorithme de typage en présence de coercions. Il est cependant désirable de donner des méthodes de sous-typage coercif dans des cadres plus généraux ainsi que des systèmes plus puissants de coercions, par exemple en présence d'une infinité de coercions.

Ce types de généralisations a déja été étudié (par exemple dans [4] et [6]) avec des résultats intéressants, mais le typage devient rapidement indécidable dès que le système de coercions devient trop expressif (voir [8]). Une extension possible de notre travail serait d'essayer d'incorporer du polymorphisme ou de la dépendance dans notre logique ou encore des types inductifs, par exemple en s'appuyant sur les travaux dans [7],[12].

Nous pensons que le point de vue du sous-typage comme une inférence logique peut potentiellement se généraliser afin d'intégrer des procédures de décision éventuellement complexes au système d'inférence de types, pour d'un coté alléger les définitions et faciliter les preuves dans un système de preuve interactif basé sur la théorie des types (Coq, Agda, Epigram...), et de l'autre faciliter la programmation dans les languages fortement typés (OCaml, Haskell...).

Un autre axe intéressant serait d'essayer de donner une extension du système de typage en présence de *types implicites* inférés à partir des arguments explicites des fonctions. Un exemple important est le problème suivant : typer le terme $eq\ x\ y$ avec eq: $\forall T$: $Type, T \to T \to Prop$, x: A, y: B et une coercion c: $A \to B$. Les algorithme de typages habituels infèrent le type T grâce au type de x, et tentent alors de typer $(eq\ A)\ x\ y$, ce qui est impossible. Pour pouvoir typer ce terme, il faudrait un système qui résolve les inéquations $A \leq T, B \leq T$ dans le système de sous-typage défini par les coercions (la solution ici serait $T = B$). Ce genre de situation semble (à notre connaissance) échapper aux études actuelles sur le sous-typage coercif.

Enfin, il serait possible d'étudier ce qui se passe dans le cas d'un typage à la Curry. L'absence d'information de types ou une information partielle peut permettre de définir une notion naturelle de sous-typage, comme celui détaillé dans [13]. La question est de savoir quelles sont les relations entre ce sous-typage et celui engendré par des coercions implicites.

Références

[1] A. R. Anderson and N. D. Belnap. *Entailment. The Logic of Relevance and Necessity, Volume 1.* U.S.A., 1975.

[2] H. Barendregt. *The Lambda Calculus : Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics.* North-Holland, revised edition, 1984.

[3] H. Barendregt. Lambda calculi with types. In S. Abramsky, D. Gabbay, and T. Maibaum, editors, *Handbook of Logic in Computer Science*, pages 117–309. Oxford Science Publications, 1992. Volume 2.

[4] V. Breazu-Tannen, T. Coquand, C. Gunter, and A. Scedrov. Inheritance as implicit coercion. 93 :172–221, 1991.

[5] G. Castagna, G. Ghelli, and G. Longo. A calculus for overloaded functions with subtyping. 117(1) :115–135, February 1995.

[6] G. Chen. Subtyping calculus of constructions. In I. Prívara and P. Ruzicka, editors, *Proceedings of MFCS'97*, volume 1295, pages 189–198, 1997.

[7] Gilles Dowek and Ying Jiang. Eigenvariables, bracketing and the decidability of positive minimal predicate logic. *Theor. Comput. Sci.*, 360(1-3) :193–208, 2006.

[8] Giorgio Ghelli. Divergence of $F_{\leq}$ type checking. *Theoretical Computer Science*, 139(1–2) :131–162, 1995.

[9] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50 :1–102, 1987.

[10] Hugo Herbelin Séquents qu'on calcule : de l'interprétation du calcul des séquents comme calcul de lambda-termes et comme calcul de stratégies gagnantes. thèse de doctorat, 1995.

[11] Longo, Milsted, and Soloviev. A logic of subtyping. In *LICS : IEEE Symposium on Logic in Computer Science*, 1995.

[12] Z. Luo. Coercive subtyping. 9(1) :105–130, February 1999.

[13] A. Miquel The Implicit Calculus of Constructions : Extending Pure Type Systems with an Intersection Type Binder and Subtyping In *Proc. of 5th Int. Conf. on Typed Lambda Calculi and Applications, TLCA'01*, Krakow, Poland, 2–5 May 2001

[14] F. Pottier. A framework for type inference with subtyping. In *Proceedings of ICFP' 98*. ACM Press, 1998.

[15] F. Henglein and J. Rehof The Complexity of Subtype Entailment for Simple Types. In *Logic in Computer Science*, pages 352-361, 1997

SAT-MICRO : petit mais costaud !

Sylvain Conchon & Johannes Kanig & Stéphane Lescuyer

LRI, Université Paris-Sud, CNRS, Orsay F-91405
INRIA Futurs, ProVal, Orsay F-91893

Résumé

Le problème SAT, qui consiste à déterminer si une formule booléenne est satisfaisable, est un des problèmes NP-complets les plus célèbres et aussi un des plus étudiés. Basés initialement sur la procédure DPLL, les *SAT-solvers* modernes ont connu des progrès spectaculaires ces dix dernières années dans leurs performances, essentiellement grâce à deux optimisations : le retour en arrière non-chronologique et l'apprentissage par analyse des clauses conflits. Nous proposons dans cet article une étude formelle du fonctionnement de ces techniques ainsi qu'une réalisation en OCAML d'un *SAT-solver*, baptisé SAT-MICRO, intégrant ces optimisations ainsi qu'une mise en forme normale conjonctive paresseuse. Le fonctionnement de SAT-MICRO est décrit par un ensemble de règles d'inférence et la taille de son code, 70 lignes au total, permet d'envisager sa certification complète.

1. Introduction

Le problème de la satisfaisabilité d'une formule propositionnelle est un des problèmes NP-complets les plus célèbres et à ce titre il a été abondamment abordé dans la littérature. Depuis quelques années, les *SAT-solvers* ont été placés au cœur de nombreuses applications industrielles, notamment grâce aux progrès spectaculaires réalisés dans leurs performances. Basés au départ sur la procédure DPLL, les *SAT-solvers* d'aujourd'hui apportent plusieurs optimisations à cet algorithme comme le retour en arrière non-chronologique, l'apprentissage par l'analyse des conflits, la détection efficace des clauses unitaires, etc. Malheureusement, il y a souvent un double fossé à combler entre les avancées théoriques et les applications pratiques : celui du passage de la formalisation à l'implémentation et celui de la preuve formelle de l'implémentation.

Nous proposons dans cet article l'étude et la réalisation d'un *SAT-solver* moderne sous un nouvel angle, de manière à répondre aux points soulevés ci-dessus. Nous présentons une implémentation en OCAML, appelée SAT-MICRO, formalisée à partir d'un système de règles d'inférence. Ces règles, très proches de l'implémentation, sont aisément adaptables aux différentes optimisations mentionnées plus haut. Bien que s'appuyant sur une architecture moderne, SAT-MICRO n'est pas aussi compétitif que les *SAT-solvers* les plus performants d'aujourd'hui. Notre objectif est ici plus de certifier une implémentation, sans sacrifier complètement l'efficacité, que de réaliser un outil hautement performant. La proximité entre la formalisation et le code, ainsi que le style de programmation purement fonctionnel de notre implémentation, font que la taille de SAT-MICRO n'excède pas 70 lignes de code et nous a permis d'entreprendre une preuve formelle de ce système.

Nous présentons en section 2 un système de règles d'inférence modélisant la procédure DPLL ainsi qu'une implémentation basée directement sur ces règles. La section 3 étend ce système avec d'une part la technique de retour en arrière non-chronologique (*backjumping)* et d'autre part un mécanisme d'apprentissage par analyse de conflits. De plus, nous montrons dans la section 4 comment adapter les *SAT-solvers* présentés dans les sections précédentes afin de manipuler des formules en forme normale conjonctive équi-satisfaisable pour éviter l'explosion combinatoire inhérente à une telle transformation. Enfin, nous montrons dans la section 5 l'impact des optimisations présentées dans la section 3 sur les performances de SAT-MICRO.

2. La procédure DPLL

L'algorithme DPLL [7, 6], nommé d'après ses inventeurs Davis, Putnam, Logemann et Loveland, est la procédure la plus classique pour décider si une formule propositionnelle en forme normale conjonctive[1] (FNC) est satisfaisable ou non. Pour cela, l'algorithme essaye de construire une instanciation des variables propositionnelles qui rende la formule vraie. En principe, l'algorithme verifie toutes les 2^n possibilités d'instancier les variables, mais il utilise deux heuristiques intelligentes qui lui permettent d'aller plus vite :

- la *propagation des contraintes booléennes* : à chaque fois qu'une valeur de vérité est choisie pour une variable, la formule est simplifiée en conséquence : les littéraux devenus faux sont supprimés des clauses, et si une clause contient un littéral devenu vrai, toute la clause est supprimée ;
- la règle de la *clause unitaire* : si la formule contient une clause formée d'un seul littéral, la valeur de vérité de la variable de ce littéral est choisie de sorte qu'il soit vrai.

De cette manière, l'algorithme procède en attribuant une valeur de vérité à chaque variable propositionnelle appartenant à la formule, jusqu'à ce que l'un des deux événements suivants arrive :

- la formule simplifiée est la conjonction vide $\emptyset$; dans ce cas, une instanciation des variables a été trouvée qui rend vraie la formule de départ : elle est donc satisfaisable et l'algorithme s'arrête ;
- l'algorithme a atteint un état dans lequel la formule simplifée contient une clause vide (un *conflit*) : dans ce cas, l'algorithme effectue un retour en arrière à l'endroit le plus récent où il peut affecter une autre valeur de vérité à une variable.

Pour la notation des formules en FNC, nous adoptons les conventions suivantes :

- l'ordre des littéraux dans une clause, ou des clauses dans une conjonction, est sans importance ; nous écrirons $l \vee C$ pour décrire une clause qui contient au moins le littéral l et $\{l_1, l_2, l_3\}$ pour la clause formée des littéraux l_1, l_2 et l_3 ;
- une formule en FNC est écrite $C_1, \ldots, C_n$ où les C_i sont les clauses de la formule. Parfois, nous omettrons les accolades des clauses singletons : si Δ est un ensemble de clauses, Δ, l est une formule avec une clause qui ne contient que le littéral l.

2.1. DPLL avec des règles d'inférences

$$\textsc{Conflict}\ \frac{}{\Gamma \vdash \Delta, \emptyset} \qquad \textsc{Assume}\ \frac{\Gamma, l \vdash \Delta}{\Gamma \vdash \Delta, l} \qquad \textsc{Bcp}\ \begin{cases} \dfrac{\Gamma, l \vdash \Delta, C}{\Gamma, l \vdash \Delta, \bar{l} \vee C} \\ \dfrac{\Gamma, l \vdash \Delta}{\Gamma, l \vdash \Delta, l \vee C} \end{cases}$$

$$\textsc{Unsat}\ \frac{\Gamma, l \vdash \Delta \qquad \Gamma, \bar{l} \vdash \Delta}{\Gamma \vdash \Delta}$$

FIG. 1 – Une version abstraite de DPLL

La figure 1 montre le fonctionnement de l'algorithme DPLL, à l'aide de cinq règles d'inférences. Elles décrivent l'état de l'algorithme par le *séquent* $\Gamma \vdash \Delta$, où Γ est l'ensemble des littéraux supposés vrais, et Δ est la formule courante. Ces règles doivent être lues de bas en haut : l'état sous le trait est l'état *avant* l'application de la règle d'inférence. La règle CONFLICT correspond au cas où l'algorithme a trouvé la clause vide dans la formule. Cette branche de l'arbre de recherche étant terminée, l'algorithme doit revenir en arrière pour trouver une autre instanciation des variables. La règle ASSUME décrit

[1] *i.e.* une formule de la forme $\bigwedge_{i=1}^{n}(l_1 \vee \cdots \vee l_{k_i})$, où chaque l_j est un littéral, c'est-à-dire une variable propositionnelle ou sa négation.

l'heuristique des clauses unitaires : un littéral dans une clause unitaire doit être supposé vrai, sans quoi la formule serait immédiatement insatisfaisable.

Les règles BCP décrivent la propagation des contraintes propositionnelles imposées par les variables dans Γ. Si un littéral est supposé faux (sa négation est dans Γ), il peut être supprimé de la clause (première règle). Si une clause contient un littéral qui est supposé vrai, la clause peut être entièrement supprimée (deuxième règle).

Finalement, la règle UNSAT, qui doit se lire de bas en haut et de gauche à droite, décrit le choix d'une valeur de vérité pour un littéral de la formule. Il est d'abord supposé vrai (partie gauche), et si aucune instanciation satisfaisable n'a été trouvée (toutes les branches de l'arbre de gauche terminent avec la règle CONFLICT), l'algorithme revient par backtracking à cette règle UNSAT et essaye la branche droite, en supposant cette fois le littéral faux.

Cette procédure a été formalisée en Coq et prouvée correcte et complète. Les tailles de la spécification et des preuves sont respectivement d'environ 400 et 2000 lignes.

2.2. Une implémentation de DPLL en OCAML

Dans cette section, nous voulons montrer qu'il n'est pas difficile de traduire les règles d'inférence de la figure 1 vers du code OCAML qui leur soit proche. Nous appelons cette première implémentation « naïve », puisqu'elle ne contient pas les optimisations que nous allons présenter dans la section 3.

Pour rendre cette implémentation indépendante de la représentation de la FNC et de la mise en FNC, nous avons besoin de deux types abstraits, un type des littéraux et un type qui représente la FNC elle-même :

```
module type LITERAL = sig
  type t
  val mk_not : t → t
  val compare : t → t → int
end
module type FNC = sig
  type formula
  module L : LITERAL
  val make : formula → L.t list list
end
```

À part le type `t` des littéraux, la signature `LITERAL` contient aussi une fonction de négation et une fonction de comparaison (nécessaire pour construire des ensembles de littéraux par le foncteur `Set.Make`). La signature `FNC` vient avec un module de type `LITERAL` et une fonction `make` qui, à partir d'une formule, construit une liste de listes de littéraux, c'est-à-dire une liste de clauses.

Nous pouvons maintenant mettre en place les structures de données pour notre *SAT-solver* naïf, tout en restant indépendant de la représentation de la FNC. Notre foncteur `Sat`, paramétré par un module du type `FNC`, déclare tout d'abord deux exceptions `Unsat` et `Sat`, le module `L` des littéraux et un module `S` d'ensembles de littéraux. Le type `t` de l'état du *SAT-solver* est tout simplement un ensemble `gamma` de littéraux et une formule en FNC `delta`, exactement comme dans les règles d'inférence de DPLL :

```
module Sat(Fnc : FNC) = struct
  exception Unsat
  exception Sat

  module L = Fnc.L
```

```
module S = Set.Make(L)
type t = { gamma : S.t; delta : L.t list list}
```

En suivant ces mêmes règles, nous pouvons procéder à l'implémentation de l'algorithme lui-même. Le code suivant implémente les règles Assume et Bcp :

```
let rec assume env f =
  if S.mem f env.gamma then env
  else bcp { gamma = S.add f env.gamma; delta = env.delta}

and bcp env =
  List.fold_left
    (fun env l → try
       let l =
         List.filter
            (fun f →
              if S.mem f env.gamma then raise Exit;
              not (S.mem (L.mk_not f) env.gamma) ) l
       in
       match l with
           [] → raise Unsat (* CONFLICT *)
         | [f] → assume env f
         | _ → {env with delta = l ::env.delta}
     with Exit → env
    ) {env with delta=[]} env.delta
```

La règle Assume se traduit aisément vers le code présenté ci-dessus. La seule différence avec la règle d'inférence est l'appel de la fonction `bcp` juste après avoir ajouté le littéral `f` dans l'environnement `gamma`. Cette fonction `bcp`, qui est bien sûr la traduction des deux règles Bcp, est plus complexe. Elle parcourt toute la formule courante (à l'aide du `fold_left`) et enlève tous les littéraux tels que leur négation est dans l'ensemble `gamma` (à l'aide du `filter`). Si le littéral lui-même apparaît dans `gamma`, toute la clause est enlevée ; ceci est implémenté à l'aide de l'exception `Exit` qui permet de sortir du filtre dès qu'un tel littéral a été trouvé. Si, au cours du parcours de la formule en FNC, on obtient la clause vide, on applique la règle Conflict et on revient en arrière en levant une exception `Unsat` (le motif de la liste vide). Si on trouve une clause unitaire (deuxième motif), le littéral correspondant est directement supposé ; les fonctions `bcp` et `assume` doivent donc être mutuellement récursives. Finalement, dans les autres cas, la clause est ajoutée à l'accumulateur du `fold_left`.

On voit bien que ces deux fonctions ne traduisent pas uniquement les règles, mais aussi une *stratégie*, c'est-à-dire une préférence de certaines règles, quand plusieurs peuvent s'appliquer. Ici, quand un littéral est supposé, les règles Bcp et Assume sont tout de suite appliquées autant que possible. Si un conflit est trouvé, la règle Conflict est appliquée. Le point fixe qui est obtenu est une formule qui ne contient plus de clause unitaire, ni de clause vide (sinon l'exception `Unsat` aurait été levée).

La seule règle qui reste à traduire est la règle Unsat.

```
let rec unsat env = try
  match env.delta with
      [] → raise Sat
    | ([_] | []) ::_ → assert false
    | (a ::_) ::_ →
        (try unsat (assume env a) with Unsat → ()) ;
         unsat (assume env (L.mk_not a))
with Unsat → ()
```

Elle se traduit par une simple analyse par cas sur la formule courante : si on trouve une FNC vide, on a trouvé une instanciation des variables qui rend vraie la formule initiale. La clause vide et une clause unitaire ne sont pas possibles ; c'est l'invariant obtenu par les fonctions `bcp` et `assume`. Sinon, un littéral quelconque `a` est choisi, et supposé vrai. Si cette branche est explorée et aucune instanciation n'a été trouvée, on suppose faux le littéral.

Aucune exception `Unsat` ne doit s'échapper de la fonction `unsat`, puisque c'est précisément ici qu'il faut la traiter et c'est pour cela qu'elle est encadrée d'un bloc `try ... with`. Pourquoi y a-t-il donc un deuxième bloc `try ... with` autour du premier appel récursif `unsat (assume env a)` ? Ce bloc est nécessaire puisqu'une exception `Unsat` peut également être lévée dans l'appel de la fonction `assume`, et dans ce cas il faut bien continuer avec le deuxième appel récursif de `unsat`. Les appels de `assume` juste avant les appels récursifs de `unsat` maintiennent l'invariant qu'il n'y a pas de clause vide ni de clause unitaire dans `delta`.

Il reste la fonction d'entrée du *SAT-solver*, que nous avons appelée `dpll`.

```
  let dpll f = try
    unsat (bcp {gamma = S.empty ; delta = Fnc.make f}) ; false
  with Sat → true | Unsat → false

end (* fin du foncteur Sat *)
```

Elle construit la FNC de la formule `f`, appelle la fonction `bcp` pour éliminer les éventuelles clauses unitaires et établir l'invariant nécessaire afin de pouvoir appeler la fonction `unsat`. Si une exception `Sat` a été levée, la formule est satisfaisable. Si l'appel de `bcp` lève `Unsat`, elle est insatisfaisable. Si la fonction `unsat` termine normalement, elle est également insatisfaisable puisque toutes les branches ont été explorées sans qu'une instanciation des variables n'ait été trouvée.

Au final, on aboutit à un code très court (environ 40 lignes pour le foncteur `Sat`) qui a l'avantage supplémentaire d'être non seulement très proche des règles d'inférence, mais aussi de la formalisation Coq : la stratégie mise en œuvre dans cette implémentation est la même que celle codée « en dur » dans la preuve de complétude.

Nous verrons par la suite comment on peut améliorer l'efficacité de cette procédure en y apportant des modifications minimales.

3. Optimisations

La procédure présentée à la section précédente reste très naïve et les *SAT-solvers* modernes, tout en s'inspirant largement de cette procédure DPLL originale, parviennent à des résultats sensiblement meilleurs grâce à un grand nombre d'optimisations [14, 9].

Un certain nombre de ces optimisations sont de nature heuristique et s'efforcent par exemple de faire les choix de littéraux de décision les plus « pertinents » possibles lorsque la règle UNSAT est appliquée. D'autres, au contraire, sont purement algorithmiques et ont pour but d'élaguer au maximum l'arbre de recherche du *SAT-solver* afin d'éviter de reproduire plusieurs fois les mêmes raisonnements au cours d'une preuve.

Dans cette section, nous allons seulement nous intéresser à ce deuxième type d'optimisations car ce sont celles qui permettent de diminuer à coup sûr la taille de l'arbre de recherche et ne se basent pas sur une quelconque propriété des formules en entrée. Nous allons en présenter deux plus en détail : le retour en arrière non chronologique et l'apprentissage par analyse des clauses conflits. Nous allons voir en particulier comment de très légères modifications du code présenté à la section précédente peuvent conduire à de singulières améliorations des performances.

3.1. Retour en arrière non chronologique

Principe. Le retour en arrière non chronologique [13] consiste, lors de l'application d'une règle UNSAT, à analyser si le littéral l introduit dans la branche de gauche a été ou non « utile » pour l'établissement du conflit dans cette branche. Dans le cas où l n'a pas servi, on peut alors se dispenser d'effectuer la recherche dans la branche de droite. Pour illustrer l'apport de cette méthode, nous avons représenté en figure 2 le fonctionnement de DPLL sur un exemple particulier.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\bar{x}_4 \vdash \{\}}}{x_3 \vdash \{\bar{x}_4\}, \{x_4\}}\,\textsc{Assume} \quad \dfrac{\dfrac{}{\bar{x}_5 \vdash \{\}}}{\bar{x}_3 \vdash \{\bar{x}_5\}, \{x_5\}}\,\textsc{Assume}}{x_2 \vdash \{\bar{x}_3, \bar{x}_4\}, \{\bar{x}_3, x_4\}, \{x_3, x_5\}, \{x_3, \bar{x}_5\}}\,\textsc{Unsat} \quad \dfrac{\vdots}{\bar{x}_2 \vdash \ldots}}{x_1 \vdash \{\bar{x}_3, \bar{x}_4\}, \{\bar{x}_3, x_4\}, \{x_2, x_3, x_5\}, \{x_3, x_5\}, \{x_3, \bar{x}_5\}}\,\textsc{Unsat} \quad \cdots}{x_0 \vdash \{\bar{x}_3, \bar{x}_4\}, \{\bar{x}_1, \bar{x}_3, x_4\}, \{x_2, x_3, x_5\}, \{x_3, x_5\}, \{x_3, \bar{x}_5\}}\,\textsc{Unsat} \quad \cdots}{\emptyset \vdash \{\bar{x}_0, \bar{x}_3, \bar{x}_4\}, \{\bar{x}_1, \bar{x}_3, x_4\}, \{x_2, x_3, x_5\}, \{x_3, x_5\}, \{x_3, \bar{x}_5\}}\,\textsc{Unsat}$$

FIG. 2 – Exemple de fonctionnement de DPLL

Seules les règles ASSUME et UNSAT y sont représentées, on suppose qu'un maximum de propagation de contraintes booléennes est réalisé après chaque application d'une de ces deux règles. Aussi, seul le dernier littéral ajouté est montré dans le contexte Γ. On peut voir dans cette figure que dans la branche où x_2 a été supposé, les conflits proviennent de l'interaction entre les littéraux x_3, x_4 et x_5. La même dérivation existe certainement dans la branche droite où $\bar{x}_2$ est supposé (marquée par des pointillés verticaux), et donc la recherche dans cette branche est effectuée inutilement par DPLL.

Modification des règles. Pour permettre au système de prendre en compte ce phénomène, il faut qu'il puisse calculer grâce à quels littéraux un conflit a été obtenu dans une branche. Pour ce faire, on modifie DPLL de la manière suivante :

- le contexte Γ contient maintenant des littéraux *annotés*, c'est-à-dire des paires $l[\mathcal{A}]$ où l est le littéral ajouté au contexte et $\mathcal{A}$ est un ensemble de littéraux appelé *dépendances* de l ; ces dépendances représentent les littéraux qui ont conduit à l'ajout de l au contexte ;
- les clauses de Δ sont elles aussi annotées par un ensemble de littéraux qui permettent de se souvenir grâce à quels littéraux les clauses ont été réduites au cours de la recherche ;
- enfin, les séquents sont maintenant de la forme $\Gamma \vdash \Delta : \mathcal{A}$ où le nouvel élément $\mathcal{A}$ est l'ensemble des littéraux nécessaires pour établir l'incompatibilité de Γ et Δ. On peut considérer ces séquents comme un algorithme prenant en entrée Γ et Δ et retournant un ensemble de littéraux $\mathcal{A}$.

$$\textsc{Conflict}\ \frac{}{\Gamma \vdash \Delta, \emptyset[\mathcal{A}] : \mathcal{A}} \qquad \textsc{Assume}\ \frac{\Gamma, l[\mathcal{B}] \vdash \Delta : \mathcal{A}}{\Gamma \vdash \Delta, l[\mathcal{B}] : \mathcal{A}} \qquad \textsc{Bcp}\ \begin{cases} \dfrac{\Gamma, l[\mathcal{B}] \vdash \Delta, C[\mathcal{B} \cup \mathcal{C}] : \mathcal{A}}{\Gamma, l[\mathcal{B}] \vdash \Delta, \bar{l} \vee C[\mathcal{C}] : \mathcal{A}} \\ \dfrac{\Gamma, l[\mathcal{B}] \vdash \Delta : \mathcal{A}}{\Gamma, l[\mathcal{B}] \vdash \Delta, l \vee C[\mathcal{C}] : \mathcal{A}} \end{cases}$$

$$\textsc{Unsat}\ \frac{\Gamma, l[l] \vdash \Delta : \mathcal{A} \qquad \Gamma, \bar{l}[\mathcal{A} \setminus l] \vdash \Delta : \mathcal{B}}{\Gamma \vdash \Delta : \mathcal{B}}\ l \in \mathcal{A} \qquad \textsc{Bj}\ \frac{\Gamma, l[l] \vdash \Delta : \mathcal{A}}{\Gamma \vdash \Delta : \mathcal{A}}\ l \notin \mathcal{A}$$

FIG. 3 – Les règles de DPLL avec retour en arrière non-chronologique

Les règles correspondant à ce mécanisme sont détaillées en figure 3. Les annotations sont introduites dans la règle UNSAT au niveau des littéraux de décision, puis elles passent naturellement des littéraux aux clauses lors de la propagation de contraintes. Lorsqu'on arrive à la règle CONFLICT, on sauve dans le membre droit du séquent l'ensemble de littéraux qui ont permis d'obtenir la clause vide et c'est cet ensemble qui permet le *backjumping* via la nouvelle règle BJ. Cette procédure a été formalisée en Coq et prouvée correcte et complète. Les tailles de la spécification et des preuves sont respectivement d'environ 450 et 3500 lignes.

Ainsi, si l'on reprend l'exemple de la figure 2, la dérivation où la règle UNSAT est appliquée avec le littéral « inutile » x_2 va maintenant utiliser la règle BJ comme représenté dans la figure 4, où $\mathcal{A}$ représente l'ensemble de littéraux $\{x_0, x_1, x_3\}$ et $\mathcal{B} = \mathcal{A} \setminus x_3 = \{x_0, x_1\}$. Puisque $\mathcal{A}$, qui décore la branche de gauche, ne contient pas x_2, la règle BJ est appliquée et la branche de droite n'est pas explorée.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\bar{x}_4[x_0,x_3] \vdash \{\}[x_0,x_1,x_3] : \mathcal{A}}\,\textsc{Conflict}}{x_3[x_3] \vdash \{\bar{x}_4\}[x_0,x_3], \{x_4\}[x_1,x_3] : \mathcal{A}}\,\textsc{Assume} \qquad \dfrac{\dfrac{}{\bar{x}_5[x_0,x_1] \vdash \{\}[x_0,x_1] : \mathcal{B}}\,\textsc{Conflict}}{\bar{x}_3[x0,x_1] \vdash \{\bar{x}_5\}[x_0,x_1], \{x_5\}[x_0,x_1] : \mathcal{B}}\,\textsc{Assume}}{x_2[x_2] \vdash \{\bar{x}_3,\bar{x}_4\}[x_0], \{\bar{x}_3,x_4\}[x_1], \{\bar{x}_4,\bar{x}_5\}[], \{x_3,x_5\}[], \{x_3,\bar{x}_5\}[] : \mathcal{B}}\,\textsc{Unsat}}{x_1[x_1] \vdash \{\bar{x}_3,\bar{x}_4\}[x_0], \{\bar{x}_3,x_4\}[x_1], \{x_2,x_3,x_5\}[], \{x_3,x_5\}[], \{x_3,\bar{x}_5\}[] : \mathcal{B}}\,\textsc{BJ}$$

FIG. 4 – Exemple de fonctionnement de DPLL avec *backjumping*

Implémentation. Par rapport à l'implémentation de DPLL donnée à la section 2, il n'y a que très peu de choses à changer pour ajouter le retour en arrière non chronologique. L'environnement `gamma` est maintenant représenté par un dictionnaire où les littéraux sont associés à leurs dépendances, tandis que `delta` est une liste de couples, les clauses étant également associées à des dépendances :

```
module L = Fnc.L
module S = Set.Make(L)
module M = Map.Make(L)

type t = { gamma : S.t M.t ; delta : (L.t list × S.t)  list }
```

Les fonctions `assume` et `bcp` fonctionnent comme précédemment, à ceci près qu'elles sont adaptées de telle sorte que la propagation des dépendances soit calculée correctement, et que l'exception `Unsat`, levée lorsqu'une clause vide est rencontrée, renvoie maintenant l'ensemble de dépendances associé à cette clause vide. La fonction `unsat` peut ainsi s'en servir pour effectuer une règle BJ ou non :

```
let rec unsat env = try
  match env.delta with
      [] → raise Sat
    | ([_],_ | [],_) ::_ → assert false
    | ((a ::_),_) ::_ →
      let d =
        try unsat (assume env (a,S.singleton a)) with Unsat d → d in
      if not (S.mem a d) then d
      else unsat (assume env (L.mk_not a,S.remove a d))
 with Unsat d → d
```

Enfin, le point d'entrée de la procédure est la fonction `dpll`, qui utilise la fonction `dispatch` pour annoter toutes les clauses avec l'ensemble vide et appelle alors la fonction `unsat` sur le séquent obtenu.

```
let dispatch d = List.map (fun l→ l,d)

let dpll f = try
  let  _ =
    unsat (bcp {gamma = M.empty ; delta = dispatch S.empty
    (Fnc.make f)}) in
  false
with Sat → true | Unsat _ → false
```

3.2. Apprentissage

Principe. Si l'ajout du retour en arrière non chronologique permet bien d'élaguer certaines parties de l'arbre de recherche à partir des conflits déjà trouvés, il ne tire pas complètement partie des informations à sa disposition. Pour s'en rendre compte, il suffit de considérer la situation schématisée par la figure 5.

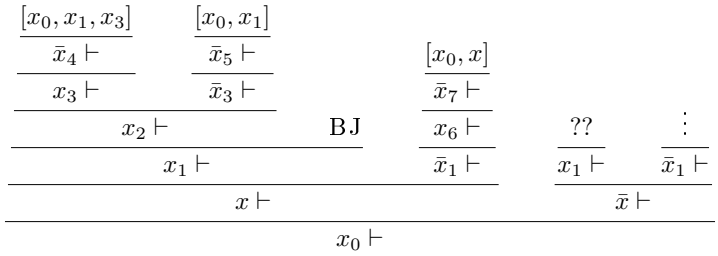


FIG. 5 – Exemple montrant l'insuffisance du *backjumping*

Cette figure schématise une dérivation dans le système de la figure 3 similaire à celle donnée en exemple en figure 2. Seuls les littéraux introduits sont représentés, ainsi que les ensembles de dépendances obtenus aux différentes feuilles. La différence entre la dérivation de la figure 4 et celle-ci est qu'ici, un autre littéral x a été introduit dans le contexte par une règle UNSAT entre les introductions des deux littéraux x_0 et x_1. Or, ce sont ces deux littéraux qui permettaient à eux deux de créer le conflit puisqu'après le *backjumping* sur x_2, les dépendances associées au séquent étaient justement l'ensemble $\mathcal{B} = \{x_0, x_1\}$.

Cela signifie qu'en particulier, supposer x_0 et x_1 vrais en même temps mène à un conflit dans ce problème. Ainsi, on sait que parcourir la branche marquée par les points d'interrogation ne servirait à rien puisque dans cette branche x_0 et x_1 sont vrais. Cependant, le retour en arrière non chronologique ne peut pas nous aider sur ce plan puisque les informations de dépendances qu'il utilise sont perdues dès lors que l'algorithme est repassé « en-dessous » d'un branchement où un des littéraux de dépendance avait été introduit. Ici, en redescendant de la branche où $\bar{x}_1$ est supposé, le nouvel ensemble de dépendances est $[x, x_0]$ et ne peut plus de toute façon faire intervenir x_1 : en revenant dans le branchement sur x, on a perdu l'information comme quoi x_0 et x_1 ne faisaient pas bon ménage et on ne peut pas l'exploiter dans la suite du parcours de l'arbre.

Modification des règles. Pour résoudre ce problème, une solution consiste à stocker, en plus de l'ensemble de dépendances courant, un ensemble de clauses appelées *clauses conflits* représentant l'ensemble des clauses que l'on a déjà « apprises » au cours de l'algorithme. Sur notre exemple, on a appris que x_0 et x_1 impliquent la clause vide, ce que l'on garde sous forme de dépendances : lorsque l'on redescend au branchement sur x, on veut faire en sorte de se souvenir que x_0 implique $\bar{x}_1$: autrement

dit, on veut passer de $\emptyset[x_0, x_1]$ à $\{\bar{x}_1\}[x_0]$. Plus généralement, nous allons considérer que nos clauses conflits sont des clauses annotées et définir une opération sur les ensembles de clauses conflits notée Shift_l et permettant de sortir un litéral l des annotations :

$$\begin{aligned}\mathit{Shift}_l(\emptyset) &= \emptyset\\ \mathit{Shift}_l(\{C[\mathcal{A}, l]\} \cup \mathbb{A}) &= \{\bar{l} \vee C[\mathcal{A}]\} \cup \mathit{Shift}_l(\mathbb{A})\\ \mathit{Shift}_l(\{C[\mathcal{A}]\} \cup \mathbb{A}) &= \{C[\mathcal{A}]\} \cup \mathit{Shift}_l(\mathbb{A}) \text{ si } l \notin \mathcal{A}\end{aligned}$$

Les séquents s'écrivent maintenant $\Gamma \vdash \Delta : \mathcal{A}, \mathbb{A}$ où le nouvel élément $\mathbb{A}$ est l'ensemble des clauses conflits. Les règles sont très semblables à celles du système de la figure 3 et n'ajoutent que le traitement des clauses conflits ; elles sont présentées dans la figure 6. Celles-ci proviennent des dépendances trouvées lors des conflits, et la règle UNSAT permet d'ajouter $\bar{l}[\mathcal{A} \setminus l]$ aux clauses conflits lorsque l'ensemble de dépendances $\mathcal{A}$ contient l. Les clauses sont maintenues par toutes les règles, à ceci près que les règles UNSAT et BJ appliquent la fonction Shift_l aux clauses conflits trouvées dans la première branche, comme nous le suggérions ci-dessus. Enfin, ces clauses sont utilisées dans la branche droite de la règle UNSAT et sont ajoutées au contexte afin d'aider à accélérer la recherche d'un conflit dans cette branche. En effet, les clauses de $\mathit{Shift}_l(\mathbb{A})$ qui contiennent $\bar{l}$ seront éliminées par BCP, ce sont les clauses restantes qui permettront éventuellement d'obtenir un conflit plus rapidement.

$$\textsc{Conflict}\ \frac{}{\Gamma \vdash \Delta, \emptyset[\mathcal{A}] : \mathcal{A}, \emptyset} \qquad \textsc{Assume}\ \frac{\Gamma, l[\mathcal{B}] \vdash \Delta : \mathcal{A}, \mathbb{A}}{\Gamma \vdash \Delta, l[\mathcal{B}] : \mathcal{A}, \mathbb{A}}$$

$$\textsc{Bcp}\ \begin{cases} \dfrac{\Gamma, l[\mathcal{B}] \vdash \Delta, C[\mathcal{B} \cup \mathcal{C}] : \mathcal{A}, \mathbb{A}}{\Gamma, l[\mathcal{B}] \vdash \Delta, \bar{l} \vee C[\mathcal{C}] : \mathcal{A}, \mathbb{A}} \\ \dfrac{\Gamma, l[\mathcal{B}] \vdash \Delta : \mathcal{A}, \mathbb{A}}{\Gamma, l[\mathcal{B}] \vdash \Delta, l \vee C[\mathcal{C}] : \mathcal{A}, \mathbb{A}} \end{cases}$$

$$\textsc{Unsat}\ \frac{\Gamma, l[l] \vdash \Delta : \mathcal{A}, \mathbb{A} \qquad \Gamma, \bar{l}[\mathcal{A} \setminus l] \vdash \Delta, \mathit{Shift}_l(\mathbb{A}) : \mathcal{B}, \mathbb{B}}{\Gamma \vdash \Delta : \mathcal{B}, \mathit{Shift}_l(\mathbb{A}) \cup \{\bar{l}[\mathcal{A} \setminus l]\} \cup \mathbb{B}}\ l \in \mathcal{A}$$

$$\textsc{Bj}\ \frac{\Gamma, l[l] \vdash \Delta : \mathcal{A}, \mathbb{A}}{\Gamma \vdash \Delta : \mathcal{A}, \mathit{Shift}_l(\mathbb{A})}\ l \notin \mathcal{A}$$

FIG. 6 – Les règles de DPLL avec apprentissage par analyse des clauses conflits

Implémentation. Cette fois encore, les modifications à apporter à l'implémentation afin de mettre en place cette optimisation sont relativement restreintes. Le contexte `gamma` et l'ensemble des clauses `delta` d'un séquent sont les mêmes qu'auparavant. Les fonctions `bcp` et `assume` ne sont pas du tout modifiées par rapport à la section précédente. Seule la fonction `unsat` est en fait modifiée : elle doit retourner non plus seulement un ensemble de dépendances comme avant, mais également un ensemble de clauses annotées.

On définit donc via le foncteur `Set.Make` le module `C` des ensembles de clauses annotées, ainsi que l'implémentation de la fonction *Shift* sur ces ensembles.

```
module C = Set.Make(
  struct
    type t = L.t list × S.t
```

```
    let compare (cl1,s1) (cl2,s2) = ...
  end)

let shift a c =
  C.fold (fun ((cl,d) as x) c →
            let x = if S.mem a d then
              ((L.mk_not a) ::cl,S.remove a d) else x in
            C.add x c ) c C.empty
```

Ensuite, la fonction `unsat` se contente d'appliquer la fonction `shift` aux clauses retournées, fait le *backjumping* si c'est possible ou bien ajoute les clauses conflits au contexte de la branche droite dans le cas contraire.

```
let rec unsat env = try
  match env.delta with
      [] → raise Sat
    | ([_],_ | [],_) ::_ → assert false
    | ((a ::_),_) ::_ →
        let d , c =
          try unsat (assume env (a,S.singleton a))
          with Unsat r → r , C.empty in
        let c = shift a c in
        if not (S.mem a d) then d , c
        else
          let (n,dn) as x = L.mk_not a,S.remove a d in
          let d , c' =
            unsat
              (assume
                 { env with delta = (C.elements c)@env.delta} x)
          in d , C.add ([n],dn) (C.union c c')
with Unsat d → d , C.empty
```

4. Formes normales conjonctives

Comme nous l'avons vu dans les sections précédentes, la procédure DPLL — et ses extensions — manipulent exclusivement des formules en forme normale conjonctive (FNC). Une transformation est donc nécessaire si l'on veut vérifier la satisfaisabilité d'une formule arbitraire à l'aide de cette procédure. Maleureusement, il n'existe pas d'algorithme polynomial de mise en forme clausale[2] et cette transformation peut donc s'avérer catastrophique en pratique. Une solution à ce problème consiste à engendrer des FNC qui ne sont plus équivalentes à la formule de départ mais seulement équi-satisfaisables.

Nous présentons dans cette section une extension de DPLL pour manipuler, de manière incrémentale, des FNC équi-satisfaisables. Nous proposons également une implémentation du module `Fnc`, basée sur la technique du *hash-consing*, pour construire ces FNC équi-satisfaisables de manière efficace.

[2]S'il en était autrement on aurait alors également, par dualité, un algorithme polynomial de mise en DNF et donc une preuve de $P = NP$.

4.1. FNC équi-satisfaisables

Afin de déterminer la satisfaisabilité d'une formule booléenne ψ quelconque, les *SAT-solvers* présentés en sections 2 et 3 réalisent une mise en forme clausale à l'aide de la fonction `Fnc.make` qui convertit ψ en une liste de clauses de type `L.t list list`, où `L.t` représente le type des littéraux.

Bien que nécessaire, cette transformation peut néanmoins constituer un véritable frein à l'efficacité de ces outils. Dans le pire des cas, la taille de la forme normale conjonctive d'une formule ψ peut être $O(2^{|\psi|})$. Afin de se faire une idée de l'explosion combinatoire de cette transformation, le résultat de la mise en FNC de la formule ψ de la forme $(a_1 \wedge a_2 \ldots \wedge a_n) \vee (b_1 \wedge b_2 \ldots \wedge b_k)$ est la formule suivante :

$$\bigwedge_{i=1}^{n} \bigwedge_{j=1}^{k} (a_i \vee b_j)$$

Une solution bien connue pour contourner cette difficulté est de construire une forme normale conjonctive ϕ de taille proportionnelle à ψ, telle que ϕ soit *seulement* équi-satisfaisable, mais non-équivalente, à ψ (voir par exemple [12, 8]). L'idée consiste à introduire de nouvelles variables X (appelées *proxies*) pour remplacer les sous-formules φ de ψ et à définir les clauses représentant l'équivalence $\varphi \leftrightarrow X$. Par exemple, on peut remplacer les sous-formules $a_1 \wedge \ldots \wedge a_n$ et $b_1 \wedge \ldots \wedge b_k$ de ψ par deux *proxies* X et Y et ajouter les clauses pour représenter les deux équivalences $X \leftrightarrow (a_1 \wedge \ldots \wedge a_n)$ et $Y \leftrightarrow (b_1 \wedge \ldots \wedge b_k)$. On obtient alors la FNC suivante :

$$(X \vee Y) \wedge \bigwedge_{i=1}^{n} (\bar{X} \vee a_i) \wedge \bigwedge_{j=1}^{k} (\bar{Y} \vee b_j) \wedge \left(X \vee \bigvee_{i=1}^{n} \bar{a}_i \right) \wedge \left(Y \vee \bigvee_{j=1}^{k} \bar{b}_j \right)$$

Bien que cette solution réduise considérablement la taille des FNC, on peut encore réduire le nombre de clauses manipulées par un *SAT-solver* en retardant l'introduction des clauses définissant les *proxies* jusqu'au moment où ces variables se voient affecter une valeur par le *SAT-solver*. Par exemple, seule la formule $X \vee Y$ de la FNC ci-dessus peut être donnée au démarrage du *SAT-solver*, les clauses définissant X et Y peuvent elles être ajoutées seulement quand ces variables se voient assigner une valeur (positive ou négative).

Nous détaillons dans la suite de cette section un algorithme de mise en forme clausale utilisant cette technique des *proxies*, ainsi qu'un mécanisme d'ajout de clauses *incrémental* qui ne nécessite qu'un changement minimal dans les codes des *SAT-solvers* présentés dans les sections précédentes.

Principe. On remarque dans la FNC précédente que les clauses qui définissent, par exemple, le *proxy* X se réduisent à la formule $\bigwedge_{i=1}^{n} a_i$ quand X est vrai et à la formule $\bigvee_{i=1}^{n} \bar{a}_i$ quand X est faux. C'est donc l'une ou l'autre de ces deux formules que l'on souhaite insérer dans le *SAT-solver* quand X se voit attribuer une valeur. La même remarque s'applique aux clauses définissant les *proxies* des sous-formules de la forme $f \vee g$, $f \rightarrow g$ et $f \leftrightarrow g$. Le tableau de la figure 7 résume la forme de ces clauses en fonction de la valeur assignée au *proxy* par le *SAT-solver*.

	X	$\bar{X}$
$X \leftrightarrow f \wedge g$	$f \wedge g$	$\bar{f} \vee \bar{g}$
$X \leftrightarrow f \vee g$	$f \vee g$	$\bar{f} \wedge \bar{g}$
$X \leftrightarrow (f \rightarrow g)$	$\bar{f} \vee g$	$f \wedge \bar{g}$
$X \leftrightarrow (f \leftrightarrow g)$	$(\bar{f} \vee g) \wedge (f \vee \bar{g})$	$(f \vee g) \wedge (\bar{f} \vee \bar{g})$

FIG. 7 – Clauses définissant les *proxies*

Modification des règles. On modifie les règles ASSUME et UNSAT de la procédure DPLL afin d'introduire dans Δ les clauses qui définissent un *proxy* au moment où cette variable est ajoutée à Γ. La figure 8 montre les modifications apportées aux règles de la figure 1. La fonction `expand` prend en argument un littéral l et retourne un ensemble de clauses. Cet ensemble doit contenir des clauses de la forme de celles présentées dans le tableau de la figure 7, si l est un *proxy*. Il doit être vide si l est un littéral « traditionnel ». Des modifications similaires peuvent être apportées aux règles de DPLL avec *backjumping* et apprentissage : il faut simplement attacher les informations de dépendances portées par l aux clauses de `expand(`l`)`.

Implémentation. Une fonction `expand` de type `L.t → L.t list list` est ajoutée à la signature `FNC`. Le type `L.t` représente donc maintenant à la fois les littéraux « traditionnels » et les *proxies*. La nature des variables est indifférente au *SAT-solver*, seule la fonction `expand` doit pouvoir les distinguer.

```
module type FNC = sig
  type formula
  module L : LITERAL
  val make : formula → L.t list list
  val expand : L.t → L.t list list
end
```

L'unique modification à apporter au *SAT-solver* de la section 2 est la concaténation à `delta` de la liste de clauses retournées par `Fnc.expand l`, au moment où le littéral `l` est passé en argument à la fonction `assume`. On obtient la fonction `assume` suivante :

```
let rec assume env l =
  if S.mem l env.gamma then env
  else bcp { gamma = S.add l env.gamma ;
             delta = (Fnc.expand l)@env.delta }
```

Une modification similaire doit être réalisée pour la fonction `assume` des *SAT-solvers* de la section 3. On utilise la fonction `dispatch` pour propager les informations de dépendances associées au littéral `l`. On obtient le code suivant :

```
let rec assume env (l,d) =
  if M.mem l env.gamma then env
  else bcp { gamma = M.add l d env.gamma ;
             delta = (dispatch d (Fnc.expand l))@env.delta}
```

4.2. Implémentation du module `Fnc`

Nous présentons brièvement dans cette section une implémentation efficace d'un module `Fnc` pour construire des FNC équi-satisfaisables. La méthode mise en œuvre repose sur la la technique de *hash-consing*. Le lecteur intéressé trouvera de plus amples détails dans [5].

$$\text{ASSUME}\ \frac{\Gamma, l \vdash \texttt{expand}(l), \Delta}{\Gamma \vdash \Delta, l} \qquad \text{UNSAT}\ \frac{\Gamma, l \vdash \texttt{expand}(l), \Delta \qquad \Gamma, \bar{l} \vdash \texttt{expand}(l), \Delta}{\Gamma \vdash \Delta}$$

FIG. 8 – Les règles de DPLL avec *proxies*

Principe. La méthode suit la structure récursive des formules. Par exemple, si ψ est une formule de la forme $f \wedge g$, on crée récursivement les *proxies* X_f et X_g de f et g, respectivement, puis on construit les deux listes de clauses ψ^+ et ψ^- en utilisant X_f et X_g, comme indiqué par le tableau de la figure 7. Maintenant, plutôt que de créer une nouvelle variable pour remplacer ψ, nous allons simplemement *hash-conser* la paire (ψ^+, ψ^-) et utiliser la valeur obtenue comme *proxy* pour ψ. Ainsi, nous réalisons non seulement une FNC équi-satisfaisable mais nous assurons également un partage maximal des sous-formules (donc des *proxies*), ce qui peut s'avérer très efficace pour les *SAT-solvers* (cf. section 5).

Implémentation. Nous utilisons la librairie `Hashcons` présentée dans [5] pour fabriquer des FNC équi-satisfaisables. Ce module contient une fonction `hashcons`, de type α `t` $\rightarrow \alpha \rightarrow \alpha$ `hash_consed`, permettant d'*hash-conser* des valeurs d'un type quelconque α. Le premier argument de la fonction est une table contenant les valeurs déjà partagées. Le résultat est une valeur dont le type, α `hash_consed`, permet de distinguer les valeurs *hash-consées* de celles qui ne le sont pas encore (très utile pour assurer le partage maximal). `hash_consed` est un type enregistrement ayant une étiquette `node` de type α dans laquelle est stockée la valeur passée à la fonction `hashcons`. Le type `L.t` des littéraux est défini à l'aide des trois types récursifs ci-dessous. Le type `t` représente les *proxies* des sous-formules. Il correspond aux paires *hash-consées* des clauses de la figure 7, représentées ici par des enregistrements de type `view` avec deux étiquettes `pos` et `neg`. Les clauses sont définies récursivement par le type `clause` qui représente soit une liste de clauses *hash-consées*, soit directement un littéral « traditionnel ».

```
module L = struct
 type t = view Hashcons.hash_consed
 and view = { pos : clauses ; neg : clauses}
 and clauses = C of t list list | LT of string × bool
```

La fonction `mk_not` construit la négation d'un *proxy* x simplement en échangeant le contenu des champs `x.node.pos` et `x.node.neg`, puis en *hash-consant* le nouvel enregistrement :

```
 let mk_not x = let n = x.node in hashcons table {pos=n.neg ;neg=n.pos}
```

La fonction `expand` du module `Fnc` se contente de retourner la liste des clauses contenues dans le champ `pos` du *proxy* `x`. Elle renvoie la liste vide si `x` représente un littéral « traditionnel » :

```
let expand x = match x.node.pos with C l → l | LT _ → []
```

Enfin, on construit le *proxy* d'une sous-formule comme $f \wedge g$ à l'aide de la fonction `mk_and` ci-dessous. Cette fonction prend en argument deux *proxies* f et g de type `L.t` et retourne le *proxy* correspondant au *hash-consing* des FNC $f \wedge g$ et $\bar{f} \vee \bar{g}$.

```
let mk_and f g = hashcons table {pos=C[[f] ;[g]] ; neg=C[[mk_not f ;mk_not g]]}
```

5. Performances

Nous détaillons dans cette section les résultats des tests de performances des *SAT-solvers* présentés dans les sections précédentes. Les formules utilisées pour effectuer ces expérimentations sont extraites du jeu de tests du concours annuel sur les *SAT-solvers* [1]. Ces formules sont déjà en forme normale conjonctive ; elles ne seront donc d'aucune utilité pour mesurer les effets des FNC équi-satisfaisables. Nous invitons le lecteur intéressé par les performances de la technique présentée en section 4 à consulter les résultats détaillés dans [5].

Le tableau ci-dessous récapitule les résultats obtenus sur des instances des trois familles de problèmes suivantes :

– AIM, instances satisfaisables du problème 3-SAT générées aléatoirement
– UF, instances 3-CNF mettant en jeu le phénomène de transition de phase
– DUBOIS, instances insatisfaisables générées aléatoirement

Les chiffres qui suivent les noms des formules de la première colonne indiquent le nombre de variables et le nombre de clauses de la FNC. Les trois colonnes suivantes donnent les temps de réponse (pour un Pentium 4 2GHz avec 512Mo de mémoire) des *SAT-solvers* présentés dans les Sections 2, 3.1 et 3.2, respectivement.

	DPLL	DPLL-B	DPLL-C
aim-50-1.6 (50,80)	4s	40ms	4ms
aim-100-2.0 (100,200)	> 10m	33s	0.3s
aim-200-2.0 (200,400)	> 10m	7m	4s
uf-125 (125,538)	22s	12s	10s
dubois (66,176)	8m30s	47s	52s

On peut constater que, sur ces exemples, les optimisations présentées en section 3 ont un impact fort sur les performances des *SAT-solvers*. Cet impact est à relativiser par le fait que notre système n'utilise aucune stratégie particulière pour sélectionner les littéraux de décision. Ainsi, il est très sensible à l'ordre des clauses et en particulier dans le cas de l'apprentissage, à l'ordre dans lequel les clauses conflit sont ajoutées. Ceci peut expliquer les différences de gain observées par exemple entre DUBOIS et AIM-200-2.0.

6. Travaux Connexes et Conclusion

Nous avons présenté une implémentation concise d'un *SAT-solver* moderne en OCaml à partir d'une description du système à base de règles d'inférence. Nous avons montré comment des optimisations permettant de diminuer l'espace de recherche peuvent être mises en œuvre en modifiant les règles ; d'autre part, comme ces règles ont été conçues avec le souci de la proximité entre formalisation et implémentation, le code du DPLL naïf a pu être facilement adapté afin de prendre en compte ces optimisations. Les comparaisons des résultats obtenus présentés dans la section 5 démontrent les effets importants de ces améliorations. Par ailleurs, l'implémentation de notre *SAT-solver* étant indépendante de la représentation des formules proprement dites, nous avons montré comment tirer profit de techniques permettant une mise en FNC efficace : ainsi, nous avons présenté comment un partage des sous-formules par *hash-consing* et un calcul paresseux de la FNC pouvaient être implémentés de manière transparente pour le *SAT-solver*.

Les travaux les plus proches de notre démarche sont ceux de Nieuwenhuis, Oliveras et Tinelli sur la formalisation de DPLL [11]. Leur système, basé sur des règles de réécriture, décrit une version de la procédure DPLL où les conditions de garde des règles sont exprimées de manière *abstraite* par des formules logiques. Bien que rendant plus aisée la preuve de correction, ce genre de formalisation rend plus difficile la réalisation pratique du solveur et *a fortiori* la preuve de son implémentation. Or, de part l'utilisation croissante de ces outils dans l'industrie, il est important de s'assurer de leur correction [15] : nous avons justement été en mesure de prouver formellement la correction et la complétude de nos deux premiers systèmes, et la preuve du système avec apprentissage est en cours.

Sat-Micro peut être largement amélioré en intégrant des optimisations comme le *restart* ou les *two-matched literals* [10], ainsi que de « bonnes » heuristiques pour le choix des littéraux de décision. La recherche des littéraux dont dépendent les conflits pourrait aussi être raffinée de telle sorte que seuls les littéraux « dominateurs » du conflit soient retournés. Ces littéraux, comme expliqué dans [2], permettent souvent un *backjumping* plus efficace que celui que nous avons présenté. Il serait enfin intéressant de voir comment adapter le système pour résoudre le problème de la satisfaisabilité modulo une théorie, afin de pouvoir l'exploiter plus efficacement au sein d'un solveur SMT tel Ergo [4, 3].

Références

[1] The international SAT Competitions web page. `http://www.satcompetition.org/`.

[2] P. Beame, H. Kautz, and A. Sabharwal. Understanding the power of clause learning. 2003.

[3] S. Conchon and E. Contejean. The Ergo automatic theorem prover. `http://ergo.lri.fr/`.

[4] S. Conchon, E. Contejean, J. Kanig, and S. Lescuyer. Lightweight Integration of the Ergo Theorem Prover inside a Proof Assistant. In J. Rushby and N. Shankar, editors, *AFM07 (Automated Formal Methods)*, 2007.

[5] S. Conchon and J.-C. Filliâtre. Type-Safe Modular Hash-Consing. In *ACM SIGPLAN Workshop on ML*, Portland, Oregon, Sept. 2006.

[6] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7) :394–397, 1962.

[7] M. Davis and H. Putnam. A computing procedure for quantification theory. *J. ACM*, 7(3) :201–215, 1960.

[8] T. B. de la Tour. Minimizing the number of clauses by renaming. In *CADE-10 : Proceedings of the tenth international conference on Automated deduction*, pages 558–572, New York, NY, USA, 1990. Springer-Verlag New York, Inc.

[9] J. W. Freeman. *Improvements to propositional satisfiability search algorithms*. PhD thesis, Philadelphia, PA, USA, 1995.

[10] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff : engineering an efficient sat solver. In *DAC '01 : Proceedings of the 38th conference on Design automation*, pages 530–535, New York, NY, USA, 2001. ACM Press.

[11] R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Abstract DPLL and abstract DPLL modulo theories. In F. Baader and A. Voronkov, editors, *Proceedings of the 11th International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR'04), Montevideo, Uruguay*, volume 3452 of *Lecture Notes in Computer Science*, pages 36–50. Springer, 2005.

[12] D. A. Plaisted and S. Greenbaum. A structure-preserving clause form translation. *J. Symb. Comput.*, 2(3) :293–304, 1986.

[13] J. P. M. Silva and K. A. Sakallah. Grasp : a new search algorithm for satisfiability. In *ICCAD '96 : Proceedings of the 1996 IEEE/ACM international conference on Computer-aided design*, pages 220–227, Washington, DC, USA, 1996. IEEE Computer Society.

[14] L. Zhang and S. Malik. The quest for efficient boolean satisfiability solvers. In *CADE-18 : Proceedings of the 18th International Conference on Automated Deduction*, pages 295–313, London, UK, 2002. Springer-Verlag.

[15] L. Zhang and S. Malik. Validating sat solvers using an independent resolution-based checker : Practical implementations and other applications. In *DATE '03 : Proceedings of the conference on Design, Automation and Test in Europe*, page 10880, Washington, DC, USA, 2003. IEEE Computer Society.

Vérification formelle du tri fonctionnel par tas Étude opérationnelle

Pascal Manoury

PPS – Université Paris Diderot
UFR d'informatique – Université Pierre et Marie Curie

La programmation fonctionnelle a su produire un certain nombre de «perles» : expressions élégantes ou astucieuses d'algorithmes performants en termes de purs calculs de valeurs. Parmi celles-ci, nous nous sommes penchés sur la formulation fonctionnelle, bien connue, de l'algorithme de *tri par tas* des éléments d'une liste (voir, par exemple [Bir96]).

L'algorithme fonctionnel de tri par tas repose sur le schéma général en deux étapes des tris par arbres :

1. construire une structure de tas à partir des éléments de la liste à trier.
2. extraire la liste triée de la structure de tas.

L'étape 1 est réalisée par itération d'une *fonction d'insertion* d'un nouvel élément dans la structure arborescente.

L'étude présentée ici ne concerne pas la correction *dénotationnelle* (le résultat est bien une liste triée) mais plutôt la correction *opérationnelle* de l'implantation qui repose sur la construction d'une structure d'*arbre équilibré*. À cet égard, l'élément nodal de l'expression fonctionnelle du tri par tas est la fonction d'insertion. C'est sur celle-ci que se concentre le travail présenté ici. Toute fonction construisant une structure d'arbre dont la racine minimise (ou maximise) les valeurs contenues dans l'arbre conviendra pour remplir la première étape du processus de tri. En revanche, seule une fonction construisant effectivement une structure d'arbre équilibré donnera la *performance algorithmique* attendue d'un tri pas tas. C'est en cela que nous distinguons la correction opérationnelle qui concerne les propriétés algorithmiques du processus d'obtention d'une valeur de la correction dénotationnelle qui ne concerne que les propriétés de la valeur obtenue.

Nous avions abordé cette question dans [Man96] au détour d'une étude plus générale de l'expression fonctionnelle d'algorithmes de tri dans le système Coq (à l'époque [Coq96]) dont la dernière partie était consacrée à la preuve de correction d'arbres binaires équilibrés. Cependant, pour ce qui est de la propriété d'équilibrage, nous avions travaillé sur structure d'arbre simplifiée – dans arbres sans étiquettes – ce qui avait simplifié la conduite de la preuve. Nous proposons aujourd'hui l'étude de la version complète de la fonction d'insertion. Nous n'avons pas connaissance d'autre étude axée sur une telle correction opérationnelle.

Cette étude s'inscrit dans le domaine de la *preuve de programme*. Nous en présentons le résultat selon deux formes :

- formulation et une preuve «à la main» du résultat de correction recherché ;
- formulation et preuve «à la machine» de ce même résultat en utilisant l'outil de preuve formelle PAF ! (voir [Bar03a]).

L'intérêt de cette double présentation est, à notre sens, de mesurer la distance entre la *rédaction* d'une preuve destinée à assurer un lecteur intuitif et cultivé de la validité d'un algorithme et le *codage* enchaînant et combinant un ensemble strictement défini de traits de langage de spécification et de commandes de construction de preuves formelles destinés à une vérification mécanique.

En effet, de même qu'il est par trop pénible de suivre le code d'un programme non commenté, *on ne lit pas un script de preuve formelle*. En revanche, si l'on dispose du guide d'une preuve «à la main» et si la distance entre rédaction et codage n'est pas infranchissable, le destinataire d'une preuve de programme, sans avoir besoin de déchiffrer ou rejouer l'ensemble du script de la preuve «à la machine», pourra se convaincre de la validité du codage des grandes étapes de celle-ci : l'énoncé des propriétés et des lemmes afférants. Le déchiffrage et la validation des étapes détaillées de la preuve n'offre plus d'intérêt : la machine s'en est chargé. On peut rapprocher cette méthode d'exposition de la démarche de spécification proposée par N. Lopez dans [Lop02] :

- spécification *pré-formelle* (notre preuve «à la main») ;
- spécification formelle (notre preuve «à la machine»).

dont l'objet est d'obtenir l'accord d'un client (*a priori* non spécialiste de tel ou tel système formel) sur la pertinence de l'énoncé d'une spécification formelle. Le système se chargera de la vérification mécanique de la correction du programme dérivé.

L'exposé de la preuve «à la machine» introduira au fur et à mesure des besoins les traits pertinents du système PAF !.

1. Éléments du problème

Arbre équilibré La performance algorithmique du tri par tas est obtenue si la structure d'arbre construite est *équilibrée*. Un arbre est *parfaitement équilibré* lorsque toutes ses feuilles sont à égale distance de la racine. Un arbre binaire parfaitement équilibré posséde un nombre de nœuds fonction de sa profondeur. Les listes à trier possèdent un nombre quelconque d'éléments, la propriété d'équilibrage requise pour l'algorithme de tri par tas est donc moins restrictive : on demandera simplement que pour toute paire de feuilles, leurs distances à la racine diffèrent au maximum de 1.

On peut donner une définition équivalente de cette propriété d'équilibrage évitant l'usage explicite de la quantification universelle (*«pour toute paire de feuilles...»*). Appelons *hauteur* la distance d'une feuille à la racine. Un arbre possède une *hauteur minimale* et une *hauteur maximale*. On dit alors qu'un arbre est équilibré lorsque sa hauteur minimale et sa hauteur maximale diffèrent au plus de 1.

Soit b un arbre binaire. Notons $hmin(b)$ sa hauteur minimale et $hmax(b)$ sa hauteur maximale. La propriété d'équilibrage voulue se formalise ainsi :

$$hmax(b) = hmin(b) \vee hmax(b) = hmin(b) + 1$$

Ce que l'on notera de façon plus conscise :

$$hmin(b) \pm hmax(b)$$

Pour un ensemble A d'étiquettes, on note $B[A]$ l'ensemble des arbres binaires étiquetés par les éléments de A. L'ensemble $WB[A]$ des arbres binaires équilibrés est alors défini par

$$WB[A] = \{b \in B[A] \mid hmin(b) \pm hmax(b)\}$$

On a que $b \in WB[A]$ est équivalent à $b \in B[A]$ et $hmin(b) \pm hmax(b)$.

Construction du tas : invariant La construction du tas à partir des éléments de la liste à trier est obtenue par itération, sur les éléments de la liste, d'une fonction d'insertion d'un nouvel élément a dans un tas b. Pour assurer la performance algorithmique il faut s'assurer que cette fonction préserve la propriété d'équilibrage de l'arbre binaire qu'elle produit. Appelons ins la fonction d'insertion. Un moyen immédiat d'atteindre notre but serait d'établir la simple propriété d'invariance

$$b \in WB[A] \Rightarrow ins(a, b) \in WB[A]$$

Mais nous verrons que cette invariance ne tient pas et qu'il faut en déterminer une plus fine.

Nous allons donc nous attacher, dans la suite de cet article, à poser une propriété invariance plus précise de la fonction d'insertion et nous verrons comment elle permet d'établir la correction opérationnelle recherchée. Nous préciserons pour cela comment caractériser le sous ensemble particulier des arbres engendrés par l'itération de la fonction d'insertion (les *arbres adéquats*). Cette caractérisation repose sur la notion de similarité de structure entre arbres (*arbres isomorphes*) ainsi que, et là est l'astuce, sur l'usage de la fonction d'insertion elle-même.

2. Rédaction «à la main»

Soit A un ensemble d'étiquettes. Soient Lf et Br deux symboles, respectivement d'arité 0 et 3. L'ensemble des arbres binaires $B[A]$ est donné par l'algèbre de termes :

- $Lf \in B[A]$;
- si $a \in A$, si b_1, b_2 $B[A]$ alors $Br(a, b_1, b_2) \in B[A]$.

Soit c une fonction binaire de comparaison des éléments de A. La fonction d'insertion ins est définie par les équations récursives conditionnelles suivantes :

$$\begin{array}{lcll} ins(a, Lf) & = & Br(a, Lf, Lf) & \\ ins(a_1, Br(a_2, b_1, b_2)) & = & Br(a_1, b_2, ins(a_2, b_1)) & \text{si } c(a_1, a_2) \\ & = & Br(a_2, b_2, ins(a_1, b_1)) & \text{sinon} \end{array}$$

2.1. Affiner l'invariant

On n'arrivera pas à démontrer directement que $\forall b \in B[A].b \in WB[A] \Rightarrow ins(b) \in WB[A]$. C'est à dire, pour tout $b \in B[A]$ $hmin(b) \pm hmax(b) \Rightarrow hmin(ins(b)) \pm hmax(ins(b))$, car cet énoncé est tout simplement faux. En effet l'arbre binaire $b = Br(a, Br(a, Lf, Lf), Lf)$ vérifie bien $hmin(b) \pm hmax(b)$ mais $ins(a, b) = Br(a, Lf, Br(a, Lf, Br(a, Lf, Lf)))$ et $hmax(ins(a, b)) = hmin(ins(a, b)) + 2$.

En fait, l'itération de la fonction ins, en prenant pour premier terme l'arbre vide Lf, détermine une suite particulière d'arbres équilibrés. Si l'on fait abstraction des étiquettes, on a le schéma d'insertion suivant :

$$\begin{array}{lcl} ins_a(Lf) & = & Br(Lf, Lf) \\ ins_a(Br(b_1, b_2)) & = & Br(b_2, ins_a(b_1)) \end{array}$$

L'itérée $ins_a^n(Lf)$ engendre la suite (voir figure ci-après) :

$$\begin{array}{lcl} b_0 & = & Lf \\ b_1 & = & Br(Lf, Lf) \\ b_2 & = & Br(Lf, Br(Lf, Lf)) \\ b_3 & = & Br(Br(Lf, Lf), Br(Lf, Lf)) \\ b_4 & = & Br(Br(Lf, Lf), Br(Lf, Br(Lf, Lf))) \\ b_5 & = & Br(Br(Lf, Br(Lf, Lf)), Br(Lf, Br(Lf, Lf))) \\ b_6 & = & Br(Br(Lf, Br(Lf, Lf)), Br(Br(Lf, Lf), Br(Lf, Lf))) \\ etc... & & \end{array}$$

En mettant de côté le premier terme b_0, on remarque que les arbres obtenus ont deux formes :

- soit $Br(b_i, b_i)$
- soit $Br(b_i, ins_a(b_i))$

pour un b_i quelconque de la suite. De cette remarque, on peut tirer une définition inductive de l'ensemble WBI des arbres engendrés par la fonction ins, partant de l'arbre vide :

- $Lf \in WBI$;

FIG. 1 – Étapes b_1 à b_5

- si $b \in WBI$ alors $Br(b, b) \in WBI$;
- si $b \in WBI$ alors $Br(b, ins_a(b)) \in WBI$.

Arbres isomorphes Nous ne voulons pas ici, comme nous l'avions fait dans [Man96], raisonner sur une telle abstraction de la fonction d'insertion. Il nous faut donc expliciter comment «faire abstraction des étiquettes» en introduisant la notion d'*arbres isomorphes* (arbres de même forme).

Notons $b_1 \approx b_2$ le fait, pour deux arbres binaires, d'être isomorphes. C'est une relation d'équivalence que l'on définit par :

- $Lf \approx Lf$;
- si $b_1 \approx b_2$ et $b_3 \approx b_4$ alors, pour tout $a_1, a_2 \in A$, $Br(a_1, b_1, b_3) \approx Br(a_2, b_2, b_4)$.

Arbres adéquats On définit alors l'ensemble $WBI[A]$ d'arbres binaires à étiquettes dans A – dont nous verrons qu'ils sont équilibrés et qu'ils sont en adéquation avec la fonction d'insertion – de la façon suivante :

- $Lf \in WBI[A]$;
- si $b_1 \in WBI[A]$, et si $(b_2 \approx b_1 \vee \forall a \in A.b_2 \approx ins(a, b_1))$ alors $Br(a, b_1, b_2) \in WBI[A]$.

Nous appelons *arbres adéquats* les éléments de $WBI[A]$.

Résultats La validation de la correction opérationnelle d'un programme fonctionnel de tri par tas repose sur les deux résultats suivants :

Théorème 1 *l'ensemble $WBI[A]$ est clos par la fonction ins, c'est-à-dire*

$$\forall b \in B[A].(b \in WBI[A] \Rightarrow \forall a \in A.ins(a, b) \in WBI[A])$$

Théorème 2 *tout élément de $WBI[A]$ est équilibré, c'est-à-dire $WBI[A] \subset WB[A]$, ou encore*

$$\forall b \in B[A].(b \in WBI[A] \Rightarrow b \in WB[A])$$

En effet, la fonction de construction du tas $HeapOfList$ par itération de la fonction d'insertion sur les éléments d'une liste est définie par récurrence sur la liste :

$$\begin{array}{lcl} HeapOfList(Nil) & = & Lf \\ HeapOfList(Cons(x, xs)) & = & ins(x, HeapOfList(xs)) \end{array}$$

La correction opérationnelle finale de la fonction de construction du tas est

$$\forall xs \in L[A].HeapOfList(xs) \in WB[A]$$

où $L[A]$ est l'ensemble des listes d'éléments de A. Comme on a que $\forall xs \in L[A].HeapOfList(xs) \in B[A]$ il suffit, en vertu du théorème 2, de montrer que

$$\forall xs \in L[A].HeapOfList(xs) \in WBI[A]$$

Ce que l'on obtient par induction sur la liste xs :

- si $xs = Nil$, on a bien $Lf \in WBI[A]$
- si $xs = Cons(x, xs)$, on a, par hypothèse d'induction $HeapOfList(xs) \in WBI[A]$. Il nous faut $HeapOfList(Cons(x, xs)) \in WBI[A]$, c'est-à-dire $ins(x, HeapOfList(xs)) \in WBI[A]$. Ce que l'on a en combinant le théorème 1 à l'hypothèse d'induction.

2.2. Lemmes

Dans la preuve de nos deux théorèmes nous ferons usage explicite de certaines propriétés concernant les rapports entre calcul de hauteur, fonction d'insertion, arbres isomorphes et arbres adéquats qu'énoncent les quatre lemmes que voici.

Lemme 1 *La fonction d'insertion fait croître la hauteur maximale d'au plus 1.*

$$\forall a \in A. \forall b \in B[A]. hmax(b) \pm hmax(ins(a, b))$$

Par induction structurelle sur b, on montre $\forall a \in A. hmax(b) \pm hmax(ins(a, b))$.

Si $b = Lf$, on a bien 0 ± 1.

Si $b = Br(a_1, b_1, b_2)$, on a, par hypothèse d'induction que $hmax(b_1) \pm hmax(ins(a, b_1))$, pour tout $a \in A$. Soit $a_0 \in A$ quelconque. Par définition de ins et de $hmax$[1], il faut montrer que

$$max(hmax(b_1), hmax(b_2)) \pm max(hmax(b_2), hmax(ins(\alpha, b_1))) \text{ avec } \alpha \in \{a_0, a_1\}$$

Suivant notre hypothèse d'induction, on peut donc raisonner par cas selon que $hmax(ins(\alpha, b_1)) = hmax(b_1)$ ou $hmax(ins(\alpha, b_1)) = hmax(b1) + 1$.

Si $hmax(ins(\alpha, b_1)) = hmax(b_1)$, il faut avoir

$$max(hmax(ins(\alpha, b_1)), hmax(b_2)) \pm max(hmax(b_2), hmax(ins(\alpha, b_1)))$$

ce qui est à l'évidence vrai.

Si $hmax(ins(\alpha, b_1)) = hmax(b_1) + 1$, il faut avoir

$$max(hmax(b_1), hmax(b_2)) \pm max(hmax(b_2), hmax(b_1) + 1)$$

On raisonne selon que $hmax(b_1) + 1 \leq hmax(b_2)$ ou non.

Si $hmax(b_1) + 1 \leq hmax(b_2)$, on a qu'alors $hmax(b_1) \leq hmax(b_2)$. Il faut donc $hmax(b_2) \pm hmax(b_2)$, ce qui est à l'évidence vrai.

Si $\neg(hmax(b_1) + 1 \leq hmax(b_2))$, un peu d'arithmétique nous donne que $hmax(b_2) \leq hmax(b_1)$. Il faut donc $hmax(b_1) \pm hmax(b_1) + 1$, ce qui est à l'évidence vrai et achève la démonstration.

Lemme 2 *Les arbres isomorphes ont bien même hauteur minimale et même hauteur maximale*

$$\forall b_1, b_2 \in B[A]. (b_1 \approx b_2 \Rightarrow hmin(b_1) = hmin(b_2))$$
$$\forall b_1, b_2 \in B[A]. (b_1 \approx b_2 \Rightarrow hmax(b_1) = hmax(b_2))$$

On obtient facilement ces deux énoncés par induction sur b_1, puis par cas sur b_2, dans le cas inductif.

Lemme 3 *La relation $\approx$ est invariante pour la fonction d'insertion*

$$\forall b_1, b_2 \in B[A]. (b_1 \approx b_2 \Rightarrow \forall a_1, a_2 \in A. (ins(a_1, b_1) \approx ins(a_2, b_2)))$$

[1] $hmax(Br(a, b_1, b_2) = 1 + max(hmax(b_1), hmax(b_2))$

Ici également, le résultat s'obtient directement par induction sur b_1, par cas sur b_2, puis analyse des cas introduits par la définition de *ins*.

Lemme 4 *L'ensemble* $WBI[A]$ *est clos par* $\approx$

$$\forall b_1, b_2 \in B[A].(b_1 \approx b_2, b_1 \in WBI[A] \Rightarrow b_2 \in WBI[A])$$

Ici encore, on raisonne par induction sur b_1, puis par cas sur b_2. Regardons un peu les étapes de cette preuve. On applique l'induction sur la formule $\forall b_2 \in B[A].(b_1 \approx b_2, b_1 \in WBI[A] \Rightarrow b_2 \in WBI[A])$.

Si $b_1 = Lf$ et si $b_2 = Lf$, le résultat est trivial.

Si $b_1 = Lf$ et $b_2 = Br(a, b_3, b_4)$, le résultat est trivialement vrai par fausseté de l'hypothèse $Lf \approx Br(a, b_3, b_4)$.

Si $b_1 = Br(a, b_3, b_4)$ et $b_2 = Lf$, le résultat est ici aussi trivialement vrai par fausseté de l'hypothèse $Br(a, b_3, b_4) \approx Lf$.

Si $b_1 = Br(a_1, b_3, b_4)$ et $b_2 = Br(a_2, b_5, b_6)$. On a, par hypothèse $b_3 \approx b_5$ et $b_4 \approx b_6$ ainsi que $b_3 \in WBI[A]$ et $b_4 \approx b_3 \vee \forall a \in A. b_4 \approx ins(a, b_3)$. On a, par hypothèse d'induction que $\forall b_2 \in B[A].(b_3 \approx b_2 \Rightarrow b_2 \in WBI[A])$. On veut (i) $b_5 \in WBI[A]$ et $(ii) b_6 \approx b_5$ ou $b_6 \approx ins(a, b_5)$, pour tout $a \in A$.

(i) est une conséquence de l'hypothèse d'induction.

(ii) s'obtient en raisonnant par cas d'après l'hypothèse $b_4 \approx b_3$ ou $b_4 \approx ins(a, b_3)$, pour tout $a \in A$.

Si $b_4 \approx b_3$, on a $b_6 \approx b_4 \approx b_3 \approx b_5$ en utilisant la transitivité et la symétrie de $\approx$.

Si $b_4 \approx ins(a, b_3)$, on a par lemme 3 que $ins(a, b_3) \approx ins(a, b_5)$; on a donc $b_6 \approx b_4 \approx ins(a, b_3) \approx ins(a, b_5)$.

Lemme technique : *si la fonction d'insertion fait croître strictement la hauteur maximale d'un arbre de* $WBI[A]$*, c'est que l'arbre était parfaitement équilibré.* Ce fait est la clé de la correction algorithmique de la fonction d'insertion. Il manifeste comment les étiquettes sont disposées de façon à «remplir» la structure de tas avant d'en accroître la hauteur. Il est donc intéressant d'en voir le détail.

Lemme 5

$$\forall a \in A. \forall b \in WBI[A].(hmax(ins(a, b)) = hmax(b) + 1 \Rightarrow hmin(b) = hmax(b))$$

Par induction sur b.

Le cas de base est trivial.

Si $b = Br(a', b_1, b_2)$, il faut montrer $min(hmin(b_1), hmin(b_2)) = max(hmax(b_1), hmax(b_2))$. On a, par hypothèse que $Br(a, b_1, b_2) \in WBI[A]$, ce qui nous donne en particulier que $b_1 \in WBI[A]$ et $b_2 \approx b_1$ ou $b_2 \approx ins(\alpha, b_1)$ avec $\alpha \in \{a, a'\}$. On raisonne par cas sur cette disjonction.

Si $b_2 \approx b_1$, le lemme 2 nous donne que $hmin(b_1) = hmin(b_2)$ et $hmax(b_1) = hmax(b_2)$. On peut donc se ramener à montrer que $hmin(b_1) = hmax(b_1)$. Ce que l'on aura par hypothèse d'induction si l'on montre que (i) $b_1 \in WBI[A]$ et (ii) $hmax(ins(a, b_1)) = hmax(b_1) + 1$.

(i) nous est donné par hypothèse.

(ii) par hypothèse, on a que $hmax(ins(a, Br(a', b_1, b_2))) = hmax(Br(a, b_1, b_2)) + 1$, c'est-à-dire $max(hmax(b_2), hmax(ins(\alpha, b_1))) = max(hmax(b_1), hmax(b_2)) + 1$. En utilisant le fait que l'on a ici $hmax(b_1) = hmax(b_2)$, on a en fait que $max(hmax(b_1), hmax(ins(\alpha, b_1))) = hmax(b_1) + 1$. Or, il est clair que $hmax(b_1) \leq hmax(ins(\alpha, b_1))$. On a donc $hmax(ins(\alpha, b_1)) = hmax(b_1) + 1$ avec $\alpha \in \{a, a'\}$.

Si $b_2 \approx ins(\alpha, b_1)$. On a par hypothèse que $hmax(ins(a, Br(a', b_1, b_2))) = hmax(Br(a, b_1, b_2)) + 1$, c'est-à-dire $max(hmax(b_2), hmax(ins(\alpha, b_1))) = max(hmax(b_1), hmax(b_2)) + 1$. Le lemme 2 nous donne que $hmax(b_2) = hmax(ins(\alpha, b_1))$ et l'on sait que $hmax(b_1) \leq hmax(ins(\alpha, b_1))$. On tire donc de notre hypothèse l'absurdité $hmax(ins(\alpha, b_1)) = hmax(ins(\alpha, b_1)) + 1$. Ce qui résoud trivialement l'examen de ce second cas et achève la démonstration.

2.3. $WBI[A]$ est clos par ins

Soit à montrer (théorème 1)

$$\forall b \in B[A].(b \in WBI[A] \Rightarrow \forall a \in A.ins(a,b) \in WBI[A])$$

Par induction structurelle sur b :

Si $b = Lf$, on veut, pour tout $a \in A$, $Br(a, Lf, Lf) \in WBI[A]$; c'est-à-dire, (i) $Lf \in WBI[A]$ et (ii) $Lf \approx Lf \vee Lf \approx ins(a, Lf)$. On a (i) par définition de $WBI[A]$ et (ii) car $Lf \approx Lf$ par définition de $\approx$.

Si $b = Br(a_0, b_1, b_2)$, on suppose $Br(a_0, b_1, b_2) \in WBI[A]$, c'est-à-dire

$$(H1)\ b_1 \in WBI[A] \text{ et } (H2)\ (b_2 \approx b_1 \vee \forall a \in A.b_2 \approx ins(a, b_1))$$

On a, par hypothèse d'induction (et $(H1)$) que $ins(a, b_1) \in WBI[A])$, pour tout $a \in A$.

Il faut montrer que $ins(a, Br(a_0, b_1, b_2)) \in WBI[A]$, c'est-à-dire $Br(\alpha_1, b_2, ins(\alpha_2, b_1)) \in WBI[A]$ avec $(\alpha_1, \alpha_2) = (a, a_0)$, si $c(a, a_0)$ ou $(\alpha_1, \alpha_2) = (a_0, a)$, sinon. On veut donc, par définition de $WBI[A]$

$$(i)\ b_2 \in WBI[A] \text{ et } (ii)\ ins(\alpha_2, b_1) \approx b_2 \vee \forall a \in A.ins(\alpha_2, b_1) \approx ins(a, b_2)$$

Ce que l'on montre en raisonnant par cas selon $(H2)$.

Si $b_2 \approx b_1$, (i) nous est donné par le lemme 4 et $(H1)$; (ii) nous est donné par le lemme 3.

Si $b_2 \approx ins(a, b_1)$ pour tout $a \in A$, (i) nous est donné par le lemme 4 ; (ii) est donné comme cas particulier de notre dernière hypothèse.

2.4. $WBI[A] \subset WB[A]$

Soit à montrer (théorème 2)

$$\forall b \in B[A].(b \in WBI[A] \Rightarrow b \in WB[A])$$

Par induction sur b.

Si $b = Lf$, on a $Lf \in WB[A]$ car $hmin(Lf) = hmax(Lf) = 0$.

Si $b = Br(a_0, b_1, b_2)$, supposons $Br(a_0, b_1, b_2) \in WBI[A]$, c'est-à-dire

$$(H1)\ b_1 \in WBI[A] \text{ et } (H2)\ b_2 \approx b_1 \vee \forall a \in A.b_2 \approx ins(a, b_1)$$

On a alors par hypothèse d'induction que $b_1 \in WB[A]$.

Montrons $Br(a_0, b_1, b_2) \in WB[A]$, c'est-à-dire $hmin(Br(a_0, b_1, b_2)) \pm hmax(Br(a_0, b_1, b_2))$ et, plus précisément

$$min(hmin(b_1), hmin(b_2)) \pm max(hmax(b_1), hmax(b_2))$$

On raisonne par cas selon $(H2)$:

Si $b_2 \approx b_1$, on a, par le lemme 2, que $hmin(b_1) = hmin(b_2)$ et $hmax(b_1) = hmax(b_2)$. On veut donc $hmin(b_1) \pm hmax(b_1)$, ce qui revient à $b_1 \in WB[A]$ qui nous est donné par hypothèse d'induction.

Si $b_2 \approx ins(a, b_1)$, pour tout $a \in A$, on a que $hmin(b_2) = hmin(ins(a, b_1))$ et $hmax(b_2) = hmax(ins(a, b_1))$. On veut donc, pour tout $a \in A$

$$min(hmin(b_1), hmin(ins(a, b_1))) \pm max(hmax(b_1), hmax(ins(a, b_1)))$$

Or on sait que $hmin(b_1) \leq hmin(ins(a, b_1))$ et que $hmax(b_1) \leq hmax(ins(a, b_1))$, pour tout $a \in A$. On veut donc

$$hmin(b_1) \pm hmax(ins(a, b_1))$$

D'aprés l'hypothèse d'induction $b_1 \in WB[A]$, on peut raisonner par cas, selon que $hmax(b_1) = hmin(b_1)$ ou $hmax(b_1) = hmin(b_1) + 1$.

Si $hmax(b_1) = hmin(b_1)$, on veut en fait $hmax(b_1) \pm hmax(ins(a, b_1))$, ce que l'on a par le lemme 1.

Si $hmax(b_1) = hmin(b_1) + 1$, suivant le lemme 1, on a qu'ou bien $hmax(ins(a, b_1)) = hmax(b_1)$, ou bien $hmax(ins(a, b_1)) = hmab(b_1) + 1$. Mais, si $hmax(ins(a, b_1)) = hmab(b_1) + 1$, notre lemme technique (lemme 5) nous donne que $hmin(b_1) = hmax(b_1)$ ce qui contredit l'hypothèse $hmax(b_1) = hmin(b_1)+1$. On a donc que $hmax(ins(a, b_1)) = hmax(b_1)$ et il faut montrer $hmin(b_1) \pm hmin(b_1)+1$ qui est vrai par définition de $\pm$.

3. Formalisation «à la machine»

Tournons nous à présent vers la transcription formalisée de la preuve des théorèmes 1 et 2 telle que nous avons pu la réaliser dans notre système. Bien entendu, il a fallu pour cela adapter la lettre de la formulation adoptée pour la preuve «à la main» pour en garder l'esprit. Par exemple, notre système n'est pas basé sur une théorie des ensembles, alors que nous avions employé ce formalisme. Nous avons également eu recours à des définitions inductives de relation ($\approx$) ou d'ensemble ($WBI[A]$), ce que nous ne pouvons réaliser dans notre système. Mais nous verrons comment contourner cette difficulté en mélant définition de fonctions récursives et *promotion booléenne*.

Note : nous entrelarderons la présentation qui suit de quelques «notes» décrivant les principaux traits du système PAF!

3.1. Type et fonction

Note : le système PAF! est dédié à la preuve de programmes ML. Son langage de spécification reprend la syntaxe ML des définitions de type de données[2] *et de fonctions. Notre système intègre donc (du moins, en partie) le* langage de programmation *ML. Un langage de programmation est une syntaxe, mais aussi une sémantique. Notre système intègre celle de ML sous la forme de sa* sémantique naturelle *([Kan87]) qui permet la réduction ou évaluation des expressions ML. Nous nous référerons au mécanisme de réduction comme à celui d'une* évaluation symbolique.

Structure d'arbre binaire Le type ML des arbres binaires qui nous servira est donné par :

```
type 'a btree = Lf  | Br of 'a * ('a btree) * ('a btree)
```

Note : les types de données, et plus généralement les informations de type, sont utilisés par le système à deux niveaux :
- *au niveau* syntaxique *comme des sortes. Le langage des termes et des formules est un langage* multi-sorté *;*
- *au niveau* logique *comme des prédicats. L'appartenance de la dénotation d'un terme* `t` *au type de données est notée comme une* assignation *de type (par exemple* `(t :'a btree)`*) mais il faut la lire comme la satisfaction d'un prédicat (*`t` *satisfait le prédicat «être un* `'a btree`*»). On parlera dans ce cas d'assignation* forte.

Les constructeurs sont introduits comme symboles fonctionnels libres, avec leur assignation de type forte. Les principes d'induction structurelle et par cas de construction sont inférés (voir [Kri90] et [Par92] pour les attendus fondamentaux et [Bar03b] pour la notion de typage «fort»).

[2] Les types enregistrements ne sont pas encore implantés.

Fonction d'insertion La formulation de la fonction d'ajout d'un élément dans un tas que nous utiliserons est celle-ci :

```
let rec ins_heap c a h =
  match h with
      Lf -> Br(a, Lf, Lf)
    | Br(a0, h1, h2) ->
        if (c a a0) then
          Br(a, h2, (ins_heap c a0 h1))
        else
          Br(a0, h2, (ins_heap c a h1))
```

où `c` est un paramètre fonctionnel, de type `'a -> 'a -> bool`, sensé donner l'ordre utilisé sur les étiquettes.

> *Note : le système infère le type attendu des fonctions à la manière de ML. Ici, par exemple,* `ins_heap : ('a -> 'a -> bool) -> 'a -> 'a btree -> 'a btree`*. Mais le système ne vérifie pas de lui même la validité de l'assignation forte de type, c'est-à-dire la totalité ou terminaison de la fonction. Si l'on ne fait rien de plus, le système retiendra cette information comme simple contrainte* syntaxique de sorte *et interdira l'application de la fonction à autre chose qu'un terme de la sorte attendue.*
>
> *Pour que l'assignation de type prenne son sens logique fort,* il faut prouver *que la fonction est totale, c'est-à-dire que, dans notre exemple :*
>
> ```
> Forall 'a:type. Forall c:'a -> 'a -> bool. Forall a:'a. Forall h:'a btree.
>  ((ins_heap c a h):'a btree)
> ```
>
> *Les quantificateurs de cet énoncé appellent quelques commentaires.*
> - `Forall 'a :type` *n'a pas d'autre sens que celui d'une indication de sorte. Il ne faut pas y voir un sens logique réel. La variable liée peut être instanciée par n'importe quelle expression syntaxiquement identifiée comme expression de type.*
> - `Forall c :'a -> 'a -> bool` *est une quantification du second ordre. L'assignation de type doit être lue au sens fort :* `c` *ne pourra être instancié que par des fonctions réputées totales.*
> - `Forall a :'a` *et* `Forall h :'a btree` *sont des quantification du premier ordre. Et ici également, l'assignation de type doit être lue au sens fort :* `a` *et* `h` *ne pourront être instanciées que par des expressions (disons* `e1` *et* `e2`*) dont on saura exhiber une* preuve *que* `(e1 : 'a)` *et* `(e2 :'a btree)`*.*

La preuve «à la main», de caractère algébrique, a recours aux équations conditionnelles qui ont servi à définir la fonction d'insertion. Dans la preuve «à la machine», le raisonnement équationnel est remplacé par l'invocation du mécanisme d'évaluation symbolique qui implante les régles de réductions de la sémantique naturelle. L'alternative `if-then-else` du deuxième cas de la définition ML de `ins_heap` est le pendant des équations conditionnelles de la définition algébrique. Les équations de la fonction *ins* données en 2 sont vérifiées par l'évaluation symbolique de la définition ML de `ins_heap`.

> *Note : la tactique* `SymEval` *implante, dans notre système, l'utilisation de l'évaluation symbolique pour construire les preuves. Elle prend un premier argument qui est un symbole fonctionnel. Elle calcule une* forme normale de tête *de l'expression déterminée par ce symbole. La tactique prend, optionnellement, un second argument (un entier) qui indique l'occurence du symbole à considérer. Sa valeur par défaut est 1.*
>
> *Si le symbole fonctionnel est celui d'une fonction définie, la tactique procède à l'expansion de la définition avant d'appliquer la réduction.*

3.2. Arbres équilibrés

Notre définition d'arbre équilibré repose sur les notions de *hauteur maximale* et *hauteur minimale*. Ces valeurs sont calculées respectivement par les deux fonctions

```
let rec hmax h =
  match h with
     Lf -> 0
  |  Br(a, h1, h2) -> S(max (hmax h1) (hmax h2))

let rec hmin b =
  match h with
     Lf -> 0
  |  Br(a, b1, b2) -> S(min (hmin b1) (hmin b2))
```

où `S` est la fonction successeur.

Des booléens aux propositions Pour définir la relation $\pm$, nous commençons par poser la *fonction booléenne*

```
let pred_or_eq n m = ((n = m) || ((S n)=m))
```

dont nous démontrons qu'elle est totalement définie. Ce que nous écrivions $n \pm m$, pour deux entiers n et m devient `‘(pred_or_eq n m)` dans notre système.

> *Note : remarquez la présence de l'accent grave (ou back quote) devant l'application de la fonction.*
> *La constante de prédicat discrètement notée* `‘` *est un symbole prédéfini du système qui promeut toute valeur booléenne au rang de valeur de vérité : si* `t` *est un terme de sorte booléenne (type* `bool`*), on peut former la formule atomique* `‘t` *dont la sémantique est*
> – `‘t` *est vrai si* `t` *s'évalue à la valeur* `true`
> – `Not ‘t` *est vrai si* `t` *s'évalue à la valeur* `false`
> *Notons que pour tout* `t` *de type* `bool`*, on n'a pas nécessairement que* `‘(t)` *est vrai ou faux. Il faut, pour pouvoir assigner une valeur de vérité à* `‘(t)`*, que l'on ait démontré que* `t : bool` *(au sens fort).*

Notre formulation en terme de fonction booléenne a l'avantage sur une formulation utilisant la disjonction logique (`‘(n = m) Or ‘((S n) = m)`) de pouvoir utiliser l'évaluation symbolique pour résoudre un certain nombre de cas triviaux : typiquement les cas de base des récurrences. La calcul propositionnel est ainsi, certe de façon limitée, réellement un calcul.

Le lemme 1 : *«la fonction d'insertion fait croître la hauteur maximale d'au plus 1»*, s'énonce alors

```
Theorem pred_or_eq_hmax_ins_heap :
  Forall 'a:type. Forall c:'a -> 'a -> bool. Forall x:'a. Forall b:'a btree.
    ‘(pred_or_eq (hmax b) (hmax (ins_heap c x b)))
```

Lorsque que nous avons donné la preuve «à la main» du lemme 1, nous avons utilisé le racourci suivant (souligné) :

> *Par définition de ins et de hmax, il faut montrer que*
>
> $$max(hmax(b_1), hmax(b_2)) \pm max(hmax(b_2), hmax(ins(\alpha, b_1))) \underline{\ avec\ \alpha \in \{a_0, a_1\}}$$

Dans un système formel tel que le notre, un tel racourci n'en est en fait pas un. Ce qui est en fait sous-jacent à l'utilisation de cette tournure est l'indépendance du calcul de la hauteur maximale vis-à-vis de la valeur des étiquettes présentes dans l'arbre. Nous avons, dans la preuve formalisée, utilisé explicitement ce fait en démontrant l'équation :

```
Theorem hmax_ins_heap_x :
  Forall 'a:type. Forall c:'a -> 'a -> bool. Forall b:'a btree.
  Forall x1:'a. Forall x2:'a.
    `((hmax (ins_heap c x1 b)) = (hmax (ins_heap c x2 b)))
```

> *Note : notre langage logique n'a pas de symbole d'égalité. Il ne connaît que l'égalité polymorphe prédéfinie de ML. Il n'y a donc pas à proprement parler d'équation dans notre système, mais des énoncés utilisant le mécanisme de promotion des booléens appliqué à l'expression du calcul de l'égalité de deux valeurs. La validité de tels «énoncés équationnels» peut être obtenue soit par simple évaluation symbolique, soit par preuve, en général, par induction.*
>
> *Le système de preuve offre ainsi deux formes correspondant au raisonnement équationnel : l'évaluation symbolique (implanté par la tactique* `SymEval`*) et l'utilisation d'énoncés équationnels (tactique* `Rewrite`*).*

Le résultat d'indépendance ci-dessus, nous permet d'établir, concernant la valeur de la hauteur maximale du résultat d'une insertion, l'énoncé équationnel suivant :

```
Theorem hmax_ins_heap_eq :
  Forall 'a:type. Forall c:'a -> 'a -> bool. Forall x:'a.
  Forall b1:'a btree. Forall b2:'a btree. Forall y:'a.
    `((hmax (ins_heap c y (Br(x,b1,b2))))
     = (S(max (hmax b2) (hmax (ins_heap c x b1)))))
```

qui joue le rôle, dans la preuve «à la machine» du lemme 1, de l'astuce du α dans notre preuve «à la main» de ce lemme.

Ensemble, fonction caractéristique, prédicat Nous avions défini l'ensemble des arbres équilibrés $WB[A]$ par schéma de compréhension. Ce qui n'est pas réalisable dans notre système. Cependant, on peut dire que la fonction booléenne `pred_or_eq` combinée aux fonctions `hmax` et `hmin` donne la *fonction caractéristique* de l'ensemble $WB[A]$. Posons cette fonction :

```
let is_wb b = (pred_or_eq (hmin b) (hmax b))
```

Le prédicat $hmin(b) \pm hmax(b)$ utilisé dans la définition ensembliste de $WB[A]$ devient, selon notre usage, `‘(is_wb b)`. Il suffit à caractériser l'appartenance d'une valeur `b` de type `'a btree` à l'ensemble $WB[A]$ si le paramètre A est lu comme le type paramètre `'a` et l'ensemble $B[A]$ comme le type paramétré `'a btree`. La formule `‘(is_wb b)` correspond donc à l'énoncé d'appartenance $b \in WB[A]$.

3.3. Arbres isomorphes

La définition inductive de la relation d'équivalence $\approx$ se transpose directement en terme de fonction booléenne.

```
let rec biso b1 b2 =
  match b1, b2 with
    (Lf, Lf) -> true
```

```
  | ((Br(_, b11, b12)), (Br(_, b21, b22))) -> (biso b11 b21) && (biso b12 b22)
  | _ -> false
```

Selon notre usage, le prédicat correspondant à la relation $\approx$ s'obtient par promotion booléenne.

Notez que la quantification universelle sur les étiquettes qui était nécessaire dans la définition de $\approx$ est ici masquée sous la non utilisation des étiquettes dans la définition de la fonction `biso` (motif universel `_` de la deuxième clause). Nous verrons tout à l'heure que ce miracle ne se produit pas toujours.

Le double lemme 2 : *«les arbres isomorphes ont bien même hauteur minimale et même hauteur maximale»* devient dans notre formalisation les deux lemmes

```
Theorem biso_hmax :
  Forall 'a: type. Forall b1: 'a btree. Forall b2: 'a btree.
    `(biso b1 b2) -> `((hmax b1) = (hmax b2))

Theorem biso_hmin :
  Forall 'a: type. Forall b1: 'a btree. Forall b2: 'a btree.
    `(biso b1 b2) -> `((hmin b1) = (hmin b2))
```

Le lemme 3 : *«la relation $\approx$ est invariante pour la fonction d'insertion»* devient

```
Theorem biso_ins_heap :
  Forall 'a:type. Forall c: 'a -> 'a -> bool.
  Forall x1:'a. Forall x2:'a.
  Forall b1: 'a btree. Forall b2: 'a btree.
   `(biso b1 b2) -> `(biso (ins_heap c x1 b1) (ins_heap c x2 b2))
```

La présence des deux étiquettes `x1` et `x2` nous donne le corrolaire

```
Theorem biso_ins_heap_x :
  Forall 'a:type. Forall c: 'a -> 'a -> bool. Forall x1:'a. Forall x2:'a.
  Forall b: 'a btree.
   `(biso (ins_heap c x1 b) (ins_heap c x2 b))
```

dont nous verrons l'utilité tout à l'heure.

> *Note : l'étude des cas induits par l'utilisation de l'alternative* `if-then-else` *dans la définition de la fonction d'insertion est réalisée, dans notre système, par l'application du théorème suivant*
>
> ```
> Forall 'a : type. Forall X : ('a -> Prop).
> Forall b : bool. Forall x : 'a. Forall y : 'a.
>   ((`(b) -> X(x)) And ((Not `(b)) -> X(y))) -> X(if b then x else y)
> ```
>
> *prouvable dans notre système. Son usage étant fréquent, nous avons codé la tactique dédiée* `IntroIf` *qui réalise cette application.*
>
> *La quantification* `Forall X : ('a -> Prop)` *est une quantification du second ordre. La constante* `Prop`*, et, par extension le type* `'a -> Prop`*, est, à l'instar de la constante* `type` *une indication syntaxique de sorte.*

3.4. Arbres adéquats

On ne peut transposer directement la définition inductive de l'ensemble $WBI[A]$ en posant, selon notre usage, une fonction caractéristique récursive équivalente. En effet, la définition que nous avons donné de $WBI[A]$ fait usage, dans sa clause inductive, d'une quantification universelle (*«pour tout* $a \in A$*»*).

Il faut donc trouver une *astuce* pour obtenir un effet sémantiquement similaire à l'usage de ce *«pour tout»*. L'effet de ce *«pour tout»* est d'*abstraire* l'étiquette a. Du point de vue des fonctions, l'effet d'abstraction est produit par l'introduction de *paramètres*. Lorsque l'on passe aux fonctions caractéristiques, pour obtenir un effet d'abstraction des étiquettes, il suffit de passer l'une d'elle en paramètre de la fonction. D'où la définition :

```
let rec ins_heap_wb c x b =
  match b with
    Lf -> true
  | Br(y,b1,b2) ->
     (ins_heap_wb c x b1) &&
     ((biso b2 b1) || (biso b2 (ins_heap c x b1)))
```

On pourrait s'assurer de l'indépendance de notre définition vis-à-vis du paramètre `x` en prouvant que

```
Forall x1 :'a. Forall x2 :'a. (ins_heap_wb c x1 b) = (ins_heap_wb c x2 b)
```

mais nous n'avons pas explicitement eu besoin de ce résultat dans la pratique où le lemme `biso_ins_heap_x` s'est avéré suffisant.

Le lemme 4 : *«l'ensemble* $WBI[A]$ *est clos par* $\approx$*»* devient

```
Theorem biso_ins_heap_wb :
  Forall 'a:type. Forall c: 'a -> 'a -> bool. Forall x:'a.
  Forall b1: 'a btree. Forall b2: 'a btree.
    `(biso b1 b2) -> `(ins_heap_wb c x b1) -> `(ins_heap_wb c x b2)
```

Note : la preuve de ce lemme, dans les cas où `b1` *est vide et* `b2` *non,* `b1` *n'est pas vide et* `b2` *si, fait appel au principe que ex falsum quod libet (régle d'absurdité intuitioniste). Notre système, dont le langage logique n'a pas de symbole pour l'absurdité, ne connaît qu'une version restreinte de ce principe : la régle*

```
Gamma |- `(false)
-----------------
    Gamma |- A
```

Dans notre cas, par exemple, une hypothèse telle que `` `(Lf = Br(a1, b3, b4)) `` *se réduit, par évaluation symbolique, à* `` `(false) ``*. Ce qui permet de conclure.*

Le lemme technique 5 : *«si la fonction d'insertion fait croître strictement la hauteur maximale d'un arbre de* $WBI[A]$*, c'est que l'arbre était parfaitement équilibré»* qui s'énonce

```
Theorem hmax_ins_heap_eq_S_hmax__hmin_eq_hmax :
  Forall 'a:type. Forall c: 'a -> 'a -> bool. Forall x:'a. Forall b: 'a btree.
    `(ins_heap_wb c x b) -> `((hmax (ins_heap  c x b)) = (S(hmax b)))
     -> `((hmin b) = (hmax b))
```

est maintenant abordable. Sa preuve suit globalement la structure de celle donnée «à la main», si ce n'est le recours au lemme d'indépendance `hmax_ins_heap_x` éludant l'utilisation du racourçi

$\alpha \in \{a, a'\}$. Également, l'usage du principe *ex falsum* de la résolution du dernier cas de la preuve «à la main» de ce lemme est implantée en utilisant explicitement la régle d'élimination de la négation de la déduction naturelle, en fait, la régle de négation droite de la *déduction libre*.

> *Note : le système de déduction de PAF ! est la déduction libre introduite par M. Parigot dans le cadre de l'interprétation algorithmique de la logique classique ([Par90]). Ce système formel a l'avantage de contenir, comme règles dérivées, aussi bien la déduction naturelle que le calcul des séquents. Il facilite l'implantation de nombre de* tournures de raisonnement *usuelles ; en particulier, le raisonnement sur les hypothèses qui permet de s'approcher du raisonnement* en avant *(top-down).*

3.5. Les théorèmes

Le théorème 1 : *«$WBI[A]$ est clos par la fonction d'insertion»* s'énonce

```
Theorem ins_heap_wb_ins_heap :
  Forall 'a:type. Forall c: 'a -> 'a -> bool. Forall x:'a. Forall b: 'a btree.
    `(ins_heap_wb c x b) -> `(ins_heap_wb c x (ins_heap c x b))
```

Si l'esprit de la preuve «à la machine» suit celui de la preuve «à la main», la lettre s'en éloigne en ceci : à l'instar de ce que nous avions fait pour le lemme 1, nous avons utilisé une astuce de notation :

> *Il faut montrer que* $ins(a, Br(a_0, b_1, b_2)) \in WBI[A]$, *c'est-à-dire* $Br(\alpha_1, b_2, ins(\alpha_2, b_1)) \in WBI[A]$ *avec* $(\alpha_1, \alpha_2) = (a, a_0)$, *si* $c(a, a_0)$ *ou* $(\alpha_1, \alpha_2) = (a_0, a)$, *sinon*.

Le recours au couple (α_1, α_2) a permis de factoriser l'examen des cas introduits par l'alternative `if-then-else` de la définition de la fonction d'insertion. L'usage d'un tel de biais dans la preuve «à la machine» doit être explicitement justifié, alors que l'on peut rester implicite dans la rédaction «à la main». La preuve «à la machine» procéde donc plutôt à l'examen assez redondant des deux cas de l'alternative. Cependant, et c'est l'avantage des preuves «à la machine», les resources du «copier/coller» et du «chercher/remplacer» allégent le poids d'une fastidieuse répétition.

Le théorème 2 : *«$WBI[A] \subset WB[A]$»* se formule simplement

```
Theorem ins_heap_wb_is_wb :
  Forall 'a:type. Forall c: 'a -> 'a -> bool. Forall x:'a. Forall b: 'a btree.
    `(ins_heap_wb c x b) -> `(is_wb b)
```

Ici encore, la preuve sur machine, outre qu'elle donne plus de détails, est proche, dans sa structure de la preuve «à la main». L'ultime cas de la preuve, comme dans le cas du lemme 5 est résolu par utilisation explicite de la régle d'élimination de la négation.

Conclusion

Nous avons dans cet article établi la *correction opérationelle* de la fonction nodale d'un programme fonctionnel de tri par tas. Nous avons donné deux formes de cette preuve de correction

- une usuelle pour les algorithmiciens, rédigée «à la main» et qui fait appel à l'étendue et la souplesse de la tradition mathématique appliquée aux problèmes d'informatique ;
- une entièrement formalisée à l'aide d'un outil de preuve implanté sur machine qui s'inscrit dans le cadre inauguré il y a 3 décennies des *logiques pour les fonctions calculables* ([Mil73]).
  Soulignons, qu'à l'instar de ce qui est fait pour le langage de programmation LISP par Boyer-Moore ([Boy79]), nous avons donné une preuve formelle et mécaniquement vérifiée de la correction d'un *code source* du langage de programmation ML (dans son dialecte OCAML).

On peut donc à juste titre parler de *heap sort* à propos de ce programme fonctionnel :
- sa correction dénotationnelle («c'est un tri...») a formellement été établie dans nombre de systèmes, pour ne citer qu'un proche : [Ano00] et [Fil04], ainsi que notre précédent [Man96].
- sa correction opérationnelle («...par arbre équilibré») a fait l'objet de ce papier[3].

Note : l'auteur remercie Pierre Letouzey *pour sa relecture...à l'issue de laquelle, il s'est empressé de transcrire notre preuve en Coq :)*

Références

[Ano00] (Anonyme) *Sorting : Axiomatizations of sorts Heap* The Coq Standard Library `http ://coq.inria.fr/library/Coq.Sorting.Heap.html`

[Bar03a] Baro S. (2003) *Conception et implémentation d'un système d'aide à la spécification et à la preuve de programmes ML* Thèse de l'université PARIS 7.

[Bar03b] Baro S. et Manoury P. (2003) *Un système X, Raisonner formellement sur les programmes ML* In Proc. JFLA `http ://jfla.inria.fr/2003/actes`.

[Bir96] Bird R. S. (1996) *Functional algorithm design* Science of computer programming, vol. 26.

[Boy79] Boyer R. et Moore J S. (1979) *Computational logic* Academic Press, New York.

[Coq96] Cornes C. *et al.* (1996) *The coq proof assistant reference manual, version 5.10* Technical report, INRIA.

[Fil04] Filliâtre J.-C. et Letouzey P. (2004) *Functors for Proofs and Programs* Proc. The European Symp. on Programming, LNCS 2986.

[Kan87] Kahn G. (1987) *Natural Semantics* Proc. of Symp. on theoretical aspects of computer science, Passau, Allemagne, LNCS 247.

[Kri90] Krivine J.-L. Parigot M. (1990) *Programming with proofs* Journal of Information Processing and Cybernetics, 26 :3.

[Lop02] Lopez N. (2002) *Spécification Formelle de Systèmes Complexes Méthodes et Techniques* Thèse de doctorat CNAM.

[Man96] Manoury P. (1996) *Preuves et Programmes. Un cas d'école : preuve de correction de programmes fonctionnels de tris dans le système Coq* Rapport IBP 96/22 – disponible en `http ://www.pps.jussieu.fr/~eleph/Recherche/raplitp1.ps.gz`

[Mil73] Milner R. (1973) *Logic for Computable Functions : description of a machine implementation* Technical Report, Stanford University.

[Par90] Parigot M. (1990) *Free Deduction : An Analysis of "Computations" in Classical Logic* in Proc. First Russian Conference on Logic Programming, LNCS 592.

[Par92] Parigot M. (1992) *Recursive programming with proofs* Theoretical Computer Science, 94 :2.

[3] Le script complet est disponible en `http ://www.pps.jussieu.fr/~eleph/Recherche`

Une axiomatique de la géométrie plane en Coq.

J.Duprat[1]

1: Université de Lyon,
LIP, ENS de Lyon, CNRS, INRIA, UCBL
36, Allée d'Italie, 69364 Lyon CEDEX 07, France
projet GALAPAGOS, ANR 2007
Jean.Duprat@ens-lyon.fr

1. Introduction.

Dans l'enseignement des mathématiques, la géométrie plane a toujours eu une place importante. Longtemps basée sur les "Elements" d'Euclide [Har02], elle a évolué vers un aspect plus formel avec l'axiomatisation de Hilbert [Hil71]. Toutefois, pour l'apprentissage du raisonnement, la géométrie "à la règle et au compas" conserve le privilège d'offrir le support visuel du tracé de figures.

Avec l'apparition des logiciels d'aide à la preuve, plusieurs travaux ont été réalisés dans le domaine de la géométrie. Proches de nos préoccupations, nous citerons les travaux de Von Plato sur la géométrie plane constructive [vP95], ceux de Dufour et Dehlinger sur la difficulté à implémenter la géométrie de Hilbert en logique intuitionniste [DDS00], ceux de Meikle et Fleuriot sur l'implémentation de l'axiomatique de Hilbert en Isabelle [MF03] et enfin ceux de Guilhot et Bertot sur la formalisation en Coq d'une géométrie de type "lycée" [BGP03]. De l'ensemble de ces travaux se dégagent plusieurs enseignements. Les raisonnements en géométrie plane font un large usage du tiers exclu qui l'éloigne beaucoup de la logique intuitionniste. L'axiomatisation de Hilbert est suffisament rigoureuse pour être traduite en un système cohérent d'axiomes pour un assistant de preuves, même si cette traduction met en évidence des imprécisions, surtout dans l'oubli de cas particuliers, mais le style très formel est mal adapté à l'étude des figures (c'était d'ailleurs voulu par Hilbert). Enfin construire une géométrie plus proche de celle du "lycée" conduit à adopter des axiomes au fur et à mesure de l'introduction des différentes notions [Gui05].

À notre connaissance, aucun de ces travaux n'a permis l'élaboration d'un outil de démonstration assistée par ordinateur dans le domaine de la géométrie plane utilisé par les professeurs de mathématiques de collège et de lycée. Or, quand on voit le succès du logiciel de constructions de figures géométriques Cabri [Cab06] auprès d'eux, nul doute qu'ils sauraient en faire bon usage. On se propose ici de poser un système d'axiomes dans l'objectif de construire un tel outil.

L'idée première est de séparer ce qui relève du raisonnement de ce qui relève de la construction de figures. Par exemple, le fait pour un point d'être situé entre deux autres points ou pour deux angles d'être congrus relève de la première catégorie. Dans la seconde, nous rangerons le fait de tracer une droite à partir de deux points distincts, de tracer un cercle étant connus son centre et son rayon ou de construire un point par intersection de deux figures. Le logiciel Coq [Coq05] distingue la sorte $Prop$ de la sorte Set, nous l'utiliserons en rangeant les premiers dans $Prop$ et les seconds dans Set. Bien que basé sur le calcul des constructions, Coq autorise l'ajout d'un axiome de tiers exclu dans $Prop$ sans introduction d'incohérence. Nous pourrons ainsi avoir des raisonnements analogues à ceux de la géométrie plane usuelle. De plus, le fait de ranger le tracé des figures dans Set permettra "l'extraction" de la construction d'une figure correspondant à un théorème de la même façon que l'on extrait un algorithme de calcul d'une preuve. Par exemple, de la preuve du théorème de Bolyai affirmant que trois points non alignés sont cocycliques on pourra extraire la construction du centre du cercle passant par trois points comme intersection de deux médiatrices.

Les constructeurs de figures seront la règle pour les droites, le compas pour les cercles et l'intersection pour les points. Les propriétés se déduiront des axiomes définissant le fait pour trois points d'être orientés dans le sens des aiguilles d'une montre ($Clockwise$), définissant la distance entre deux points et l'angle de droites. La propriété $Clockwise$ avait déjà été choisie par Knuth [Knu91] pour l'étude des enveloppes convexes mais son axiomatique était insuffisante pour engendrer exactement la géométrie plane. On montre que le système ainsi posé permet de retrouver toute l'axiomatique de Hilbert, à l'exception bien évidemment de l'axiome de continuité. Cette démonstration a été entièrement rédigée en Coq et ajoutée aux contributions (http ://coq.inria.fr/contribs/geometry.html).

L'article se présente comme suit. L'axiomatique du plan est décrite au chapitre 2 (orientation) et au chapitre 3 (mesures). Les constructions sont explicitées au chapitre 4. Le chapitre 5 s'attache à justifier cette construction avec l'aide de l'assistant de preuves Coq. Enfin, le chapitre 6 décrit les travaux qu'il reste à accomplir pour développer cet outil pédagogique : d'une part, l'écriture d'une bibliothèque de tactiques correspondants aux actions et assertions élémentaires dans le développement d'une preuve papier en géométrie au lycée, d'autre part une interface avec l'utilisateur et avec un logiciel de dessin de figures.

2. Les propriétés du plan.

2.1. Le plan.

Le plan est un ensemble de points, contenant au moins deux points distincts O et U.

Axiome 1.1 $\exists\, O \in Plan$,

Axiome 1.2 $\exists\, U \in Plan$,

Axiome 1.3 $O \neq U$.

Une figure du plan est définie par une propriété caractéristique vérifiée par les points de la figure. On dira qu'un point appartient à la figure ou que la figure contient (ou passe) par le point. Deux figures sont dites superposées si elles contiennent les mêmes points.

2.2. L'orientation.

On définit une propriété sur les triplets de points appelée $Clockwise$ par les axiomes suivants. Lorsqu'un triplet (A, B, C) vérifie la relation $Clockwise$, on dira que C est à droite de (A, B) et on notera $\circlearrowright\ ABC$. On choisit de définir une relation stricte excluant des triplets de points non distincts deux à deux ou des points alignés.

Axiome 2.1 l'orientation est stable par permutation circulaire des points,

$$\forall\ A\ B\ C,\ \circlearrowright\ ABC \Rightarrow \circlearrowright\ BCA$$

Axiome 2.2 l'orientation est modifiée par une transposition, les orientations dans le sens direct de ABC et de BAC s'excluent ; il est possible de n'avoir ni une orientation, ni l'autre, ce sera le cas lorsque les trois points seront alignés,

$$\forall\ A\ B\ C,\ \neg\ \circlearrowright\ ABC \vee\ \neg\ \circlearrowright\ BAC$$

Axiome 2.3 tout triplet de points (A, B, C) satisfait l'un des quatre cas :

1. C est à droite de (A, B),
2. C est à droite de (B, A),
3. tout point M à droite de (A, B) est à droite de (A, C),
4. tout point M à droite de (B, A) est à droite de (B, C) :

$$\forall\ A\ B\ C,\ \circlearrowleft ABC \vee\ \circlearrowleft BAC\ \vee\ (\forall\ M,\ \circlearrowleft ABM \Rightarrow \circlearrowleft ACM)\ \vee\ (\forall\ M,\ \circlearrowleft BAM \Rightarrow \circlearrowleft BCM).$$

Axiome 2.4 pour tout triplet (A, B, C) vérifiant $Clockwise$, tout point M est à droite de (A, B), (B, C) ou (C, A) :

$$\forall\ A\ B\ C,\ \circlearrowleft ABC \Rightarrow\ (\forall\ M,\ \circlearrowleft ABM \vee\ \circlearrowleft BCM \vee\ \circlearrowleft CAM).$$

Axiome 2.5 si tout point M à droite de (A, B) est à droite de (A, C) alors tout point M à droite de (B, A) est à droite de (C, A) :

$$\forall\ A\ B\ C,\ (\forall\ M,\ \circlearrowleft ABM \Rightarrow \circlearrowleft ACM) \Rightarrow\ (\forall\ M,\ \circlearrowleft BAM \Rightarrow \circlearrowleft CAM).$$

Axiome 2.6 si tout point M à droite de (A, B) est à droite de (C, D) et si $A \neq B$ alors tout point M à droite de (D, C) est à droite de (B, A) :

$$\forall\ A\ B\ C\ D,\ A \neq B \Rightarrow\ (\forall\ M,\ \circlearrowleft ABM \Rightarrow \circlearrowleft CDM) \Rightarrow\ (\forall\ M,\ \circlearrowleft DCM \Rightarrow \circlearrowleft BAM).$$

2.3. Quelques définitions.

L'orientation permet de définir plusieurs propriétés usuelles des points du plan.

2.3.1. Le demi-plan.

Le demi-plan à droite de (A, B) est la figure formée par les points M tels que (A, B, M) vérifie la relation $Clockwise$. Le demi-plan ainsi défini est ouvert. On remarque que le demi-plan à droite de (A, A) est vide grâce à l'axiome 2.2.
Le demi-plan à gauche de (A, B) n'a pas besoin d'être défini car c'est le demi-plan à droite de (B, A).

2.3.2. La même orientation.

Le couple (A, B) a la même orientation que le couple (C, D) si le demi-plan à droite de (A, B) est contenu dans le demi-plan à droite de (C, D). Dans cette relation, les couples (A, B) et (C, D) ont la même direction et le même sens, mais, en plus C et D ne sont pas à droite de (A, B). Pour tous couples (A, B) et (C, D) tels que (A, B) a la même orientation que (C, D) si $A = B$ alors C et D sont quelconques mais si $A \neq B$ alors $C \neq D$. La relation n'est pas symétrique.

2.3.3. La même direction.

Les couples (A, B) et (C, D) ont même direction si l'un des couples (A, B) ou (B, A) a même orientation que l'un des couples (C, D) ou (D, C) ou l'inverse.

2.3.4. La demi-droite.

La demi-droite $]A, B)$ est la figure formée des points M tels que (A, B) a même orientation que (A, M). Lorsque $A = B$, cet ensemble est le plan tout entier.
Les deux derniers cas de l'axiome 2.3 peuvent se lire :

3. C est sur la demi-droite $]A, B)$,
4. C est sur la demi-droite $]B, A)$.

2.3.5. Colinéarité.

Le point C est dit colinéaire aux points A et B s'il n'est ni à droite, ni à gauche de (A, B). Grâce à l'axiome 2.3, C est colinéaire à A et B si C appartient à la demi-droite $]A, B)$ ou à la demi-droite $]B, A)$.

2.3.6. La relation "entre".

Le point C est situé entre les points A et B s'il est distinct de A et si (A, C) et (C, B) ont la même orientation, on note $A - C - B$. Cette définition correspond à l'appartenance à l'intervalle ouvert $]A, B[$. Paradoxalement, l'appartenance de C à l'intervalle fermé $[A, B]$ s'exprime par le fait que B appartienne à la demi-droite $]A, C)$ et que A appartienne à la demi-droite $]B, C)$. Pour s'en convaincre, il suffit d'examiner tous les cas : $A - C - B$, $A = C \neq B$, $A \neq B = C$, $A = B = C$ lorsque $C \in [A, B]$, $C - A - B$, $A - B - C$, $A = B \neq C$ lorsque $C \notin [A, B]$.

3. Les mesures du plan.

3.1. La distance.

Il existe un ensemble des longueurs du plan $\mathbb{L}$ muni d'une loi de composition interne notée $+$ et d'une relation notée $<$ et une application des couples de points dans cet ensemble appelée distance et notée $(A, B) \mapsto AB$ vérifiant les axiomes suivants :

Axiome 3.1 Toutes les distances d'un point à lui-même sont égales.

$$\forall\ A\ B,\ AA = BB$$

Axiome 3.2 La distance est symétrique.

$$\forall\ A\ B,\ AB = BA$$

Axiome 3.3 : relation de Chasles Pour tout point B appartenant à l'intervalle fermé $[A, C]$, la distance AC est égale à la somme des distances AB et BC.

$$\forall\ A\ B\ C,\ B \in [A, C] \Rightarrow AC = AB + BC$$

Axiome 3.4 : réciproque de la relation de Chasles Pour tous points A, B et C tels que la distance AC est égale à la somme des distances AB et BC, le point B appartient à l'intervalle fermé $[A, C]$.

$$\forall\ A\ B\ C,\ AC = AB + BC \Rightarrow B \in [A, C]$$

Axiome 3.5 La relation $<$ est compatible avec la relation $entre$.

$$\forall\ A\ B\ C,\ B \neq C \wedge (A, B) \text{ même orientation } (B, C) \Rightarrow AB < AC$$

Axiome 3.6 Réciproquement, la relation $entre$ est compatible avec la relation $<$.

$$\forall\ A\ B\ C,\ C \in [AB) \wedge AB < AC \Rightarrow B \neq C \wedge (A, B) \text{ même orientation } (B, C)$$

Axiome 3.7 : inégalité triangulaire Si A, B et C sont trois points non alignés alors $AC < AB + BC$.

$$\forall\ A\ B\ C,\ \circlearrowleft ABC \Rightarrow AC < AB + BC$$

Axiome 3.8 La distance est archimédienne. On définit le produit externe d'une distance par un naturel n comme l'itéré de la somme de cette distance par elle-même n fois.

$$\forall\ A\ B\ C,\ A \neq B \Rightarrow \exists\ n \in \mathbb{N},\ AC < n.AB$$

3.2. Remarques sur la distance.

L'ensemble $\mathbb{L}$ va être isomorphe à l'extension de $\mathbb{Q}^+$ par la racine carrée. Toutefois, son caractère numérique n'est pas nécessaire dans une géométrie à la règle et au compas.

Par définition, on note $O_{\mathbb{L}}$ la distance OO. D'après l'axiome 3.1, toute distance AA est égale à $O_{\mathbb{L}}$. Son caractère d'élément neutre découle de la relation de Chasles.

La relation de Chasles est la véritable définition de l'addition des longueurs. On pourrait choisir comme axiome que l'aboutement des longueurs est indépendant de la droite choisie comme support et définir l'opération d'addition à partir de l'aboutement.

De même, la relation $entre$ induit la relation $<$, on pourrait choisir comme axiome que le report de longueurs sur n'importe quel support ordonne toujours les extrémités dans la même relation $entre$ et en déduire une définition de $<$.

L'énoncé textuel de l'axiome 3.5 illustre l'idée exprimée par cet axiome, mais sa formulation est plus générale. En effet, la relation $entre$ est définie (§2.3.6) par : le point B est situé entre les points A et C s'il est distinct de A et si (A, B) et (B, C) ont la même orientation. L'hypothèse de l'axiome 3.5 est : le point B est distinct de C et (A, B) et (B, C) ont la même orientation. Cette hypothèse ajoute le cas $A = B$ à celui où B est entre A et C. Ce cas particulier permettra de démontrer que $O_{\mathbb{L}}$ est la plus petite distance.

La recherche d'hypothèses minimales dans l'axiome 3.6 conduit à choisir $C \in [AB)$ pour exprimer le fait que les trois points sont alignés et que A n'est pas entre B et C. C'est l'hypothèse $AB < AC$ qui imposera $A \neq C$.

De même, l'hypothèse $\circlearrowleft ABC$ suffit dans l'axiome 3.7 à couvrir les cas A, B et C non alignés. Si les points A, B et C sont orientés dans l'autre sens, alors l'hypothèse $\circlearrowleft CBA$ impliquera la même inégalité triangulaire à la symétrie de la distance près.

3.3. Les angles.

Un angle est défini par un point, le sommet A et deux demi-droites issues de A, les côtés $]AB)$ et $]AC)$, l'angle n'est bien défini que si $A \neq B$ et $A \neq C$. Toutefois, afin d'alléger l'écriture, un angle est défini pour tout triplet et seules les manipulations exigent $A \neq B$ et $A \neq C$.

Il existe un ensemble des angles $\mathbb{A}$ et une application qui associe un élément de $\mathbb{A}$ à tout triplet de points, cette application est notée : $(A, B, C) \mapsto \widehat{ABC}$, elle vérifie les axiomes suivants :

Axiome 4.1 L'angle est un angle de demi-droites, il est indépendant de l'ordre des côtés et du point choisi sur chaque côté.

$$\forall\ A\ B\ C\ D\ E,\ A \neq B \wedge A \neq C \wedge D \in\ ]AB) \wedge E \in\ ]AC) \Rightarrow \widehat{BAC} = \widehat{EAD},$$

Axiome 4.2 : SAS Deux triangles ayant deux côtés et l'angle compris entre ces deux côtés respectivement égaux ont leurs troisièmes côtés égaux.

$$\forall\ A\ B\ C\ D\ E\ F,\ A \neq B \wedge A \neq C \wedge AB = DE \wedge AC = DF \wedge \widehat{BAC} = \widehat{EDF} \Rightarrow BC = EF,$$

Axiome 4.3 : SSS Deux triangles ayant leurs trois côtés égaux ont leurs angles respectifs égaux.

$$\forall\ A\ B\ C\ D\ E\ F,\ A \neq B \wedge A \neq C \wedge AB = DE \wedge AC = DF \wedge BC = EF \Rightarrow \widehat{BAC} = \widehat{EDF},$$

Axiome 4.4 Dans un triangle, lorsque l'on reporte les deux angles de part et d'autre du troisième sommet, les côtés de ces angles reportés sont opposés.

$$\forall\ A\ B\ C\ D\ E,\ \circlearrowleft ABC \wedge\ \circlearrowleft ACD \wedge\ \circlearrowleft ADE\ \wedge\ \widehat{BAC} = \widehat{ACD}\ \wedge\ \widehat{DAE} = \widehat{ADC}$$
$$\Rightarrow B - A - E.$$

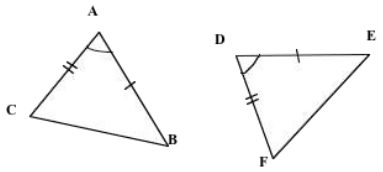


FIG. 1 – **Illustration de l'axiome 4.2.**

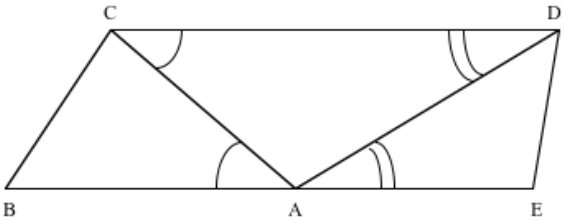


FIG. 2 – **Illustration de l'axiome 4.4.**

3.4. Remarques sur les angles.

L'axiome 4.2 est souvent appelé premier cas d'égalité des triangles. En choisissant des segments de longueur unitaire (longueur OU), on pourrait l'utiliser pour définir une mesure des angles par une longueur (deux fois le sinus de l'arc moitié). De même que pour les longueurs, le choix de postuler a priori l'existence de l'ensemble $\mathbb{A}$ simplifie la formalisation grâce à l'égalité dans $\mathbb{A}$.

L'axiome 4.3 est appelé troisième cas d'égalité des triangles. Il permet de transporter un angle à la règle et au compas.

L'axiome 4.4 s'énonce généralement sous la forme : la somme des angles d'un triangle est un angle plat, mais nous avons choisi de ne pas faire apparaître dès les axiomes la somme de deux angles. Cet axiome est nécessaire pour avoir une géométrie euclidienne dans laquelle, par un point, on ne peut faire passer qu'une parallèle à une droite donnée.

4. Les constructions du plan.

Le code Coq de ces constructions est donné dès maintenant pour expliciter les définitions.

4.1. Les droites.

4.1.1. Définition.

La droite est un type dépendant d'une figure, défini inductivement :

```
Inductive Line  : Figure -> Type :=
| Ruler : forall A B : Point, A <> B -> Line (Collinear A B)
| SuperimposedLine : forall F1 F2 : Figure,
                            Superimposed F1 F2 -> Line F1 -> Line F2.
```

Le cas de base est obtenu par le constructeur *règle* qui, à partir de deux points A et B distincts, construit la droite représentant la figure formée des points colinéaires à A et B.

Le cas inductif permet de construire la droite associée à une figure superposée à une figure représentée par une droite. Par exemple, définissons la médiatrice de deux points distincts A et B comme étant la figure formée par les points équidistants à A et B. Construisons les deux triangles équilatéraux ABC et ABD de part et d'autre de (AB). La preuve que la figure des points colinéaires à C et D est superposée à la médiatrice de A et B suffit à affirmer que cette médiatrice est une droite que l'on peut construire à partir des points C et D.

La coïncidence entre l'ensemble des points colinéaires à A et B et la droite tracée avec une règle passant par A et B est en effet un postulat de base de la géométrie à la règle et au compas.

4.1.2. Points de construction.

La définition d'une droite, en remontant la chaîne des inductions jusqu'au cas de base, permet de définir les deux points A et B qui ont servi à construire cette droite avec la *règle*. Ces points sont appelés points de construction de la droite. On écrira alors la droite : (AB). De plus, la définition d'une droite fournit une preuve que les points de construction sont distincts.

4.1.3. Propriétés des droites.

Deux droites (AB) et (CD) sont parallèles si les couples de points de constructions (A, B) et (C, D) ont la même direction. On écrira $(AB) \parallel (CD)$.

Deux droites (AB) et (CD) sont sécantes si elles ne sont pas parallèles. On écrira $(AB) \nparallel (CD)$.

4.2. Les cercles.

4.2.1. Définition.

Le cercle est un type dépendant d'une figure, défini inductivement :

```
Inductive Circle : Figure -> Type :=
| Compass : forall C A B : Point, A <> B ->
              Circle (fun M : Point => (Distance C M) = (Distance A B))
| SuperimposedCircle : forall F1 F2 : Figure,
                         Superimposed F1 F2 -> Circle F1 -> Circle F2.
```

Le cas de base est obtenu par le constructeur *compas* qui, à partir de trois points A, B et C, A et B étant distincts, construit le cercle représentant la figure formée des points dont la distance à C est égale à AB.

Le cas inductif permet de construire le cercle associé à une figure superposée à une figure représentée par un cercle.

4.2.2. Points de construction.

La définition d'un cercle, en remontant la chaîne des inductions jusqu'au cas de base, permet de définir le point C et les deux points A et B qui ont servi à construire ce cercle avec le *compas*. Ces points sont appelés points de construction du cercle, le point C est appelé *centre du cercle* et la distance AB est appelée *rayon du cercle*.

De plus, la définition d'un cercle fournit une preuve que le rayon est non nul. Le cas dégénéré du cercle de rayon nul n'a pas été retenu car il aurait introduit de nombreux cas particuliers dans les intersections.

4.2.3. Propriétés des cercles.

Deux cercles de centre C et C' et de rayon AB et $A'B'$ sont sécants si les distances CC', AB et $A'B'$ respectent la spécification du triangle, c'est-à-dire les trois inégalités triangulaires :

$$(CC' < AB + A'B') \wedge (AB < A'B' + CC') \wedge (A'B' < CC' + AB).$$

Une droite représentant la figure (D) est un diamètre du cercle de centre C si elle passe par C, c'est-à-dire si $C \in (D)$.

4.3. Les intersections.

On va donner une définition plus restrictive de l'intersection que la notion ensembliste. On ne définira que les intersections de droites et cercles permettant de construire effectivement des points du plan.

4.3.1. Définitions.

L'intersection est un type dépendant d'une figure, paramétré par deux figures $F1$ et $F2$, défini par cas :

intersection de deux droites : si les figures $F1$ et $F2$ sont représentées par deux droites sécantes, *l'intersection de deux droites* construit la représentation de la figure formée des points communs aux deux droites,

intersection de deux cercles : si les figures $F1$ et $F2$ sont représentées par deux cercles sécants, *l'intersection à droite de deux cercles* construit la représentation de la figure formée des points communs aux deux cercles situés à droite de la droite des centres, *l'intersection à gauche de deux cercles* construit la représentation de la figure formée des points communs aux deux cercles situés à gauche de la droite des centres,

intersection d'un cercle et d'un diamètre : si la figure $F1$ est représentée par un cercle et la figure $F2$ est représentée par une droite et si cette droite est un diamètre du cercle, *l'intersection positive du cercle et de son diamètre* construit la représentation de la figure formée des points communs au cercle et à la droite situés dans la direction des points de constructions de la droite par rapport au centre du cercle, *l'intersection négative du cercle et de son diamètre* construit la représentation de la figure formée des points communs au cercle et à la droite situés dans la direction opposée aux points de constructions de la droite par rapport au centre du cercle.

4.3.2. Code *Coq* de cette définition.

```
Inductive Intersection (F1 F2 : Figure) : Figure -> Type :=

  | InterLines : forall (D1 : Line F1) (D2 : Line F2),
    SecantLines F1 F2 D1 D2 ->
    Intersection F1 F2 (fun M : Point => F1 M /\ F2 M)

  | InterCirclesClock : forall (G1 : Circle F1) (G2 : Circle F2),
    SecantCircles F1 F2 G1 G2 ->
    Intersection F1 F2 (fun M : Point => F1 M /\ F2 M /\
                   Clockwise (Center F1 G1) (Center F2 G2) M)

  | InterCirclesAntiClock : forall (G1 : Circle F1) (G2 : Circle F2),
    SecantCircles F1 F2 G1 G2 ->
    Intersection F1 F2 (fun M : Point => F1 M /\ F2 M /\
```

```
                Clockwise (Center F2 G2) (Center F1 G1) M)

  | InterDiameterCirclePos : forall (D : Line F1) (G : Circle F2),
    SecantDiameterCircle F1 F2 D G ->
    Intersection F1 F2 (fun M : Point => F1 M /\ F2 M /\
                EquiOriented (LineA F1 D) (LineB F1 D) (Center F2 G) M)

  | InterDiameterCircleNeg : forall (D : Line F1) (G : Circle F2),
    SecantDiameterCircle F1 F2 D G ->
    Intersection F1 F2 (fun M : Point => F1 M /\ F2 M /\
                EquiOriented (LineB F1 D) (LineA F1 D) (Center F2 G) M).
```

4.3.3. Points d'intersection.

L'axiome de construction des points à la règle et au compas affirme que chacune des intersections définies ci-dessus construit un point unique du plan.

Axiome 5.1 il existe une application associant un unique point à chaque intersection.

$$\forall\ F1\ F2\ F, Intersection\ F1\ F2\ F\ \Rightarrow \exists!\ M \in F.$$

Cet axiome s'écrit en Coq, ($Unicity\ M\ F$ est le prédicat "tout point de F est égal au point M") :

```
Axiom PointDef : forall F1 F2 F : Figure,
  Intersection F1 F2 F -> {M : Point | F M /\ Unicity M F}.
```

4.3.4. Remarques

Cet axiome permet de construire les points à la règle et au compas satisfaisant la propriété de la figure F. De plus, il permet de démontrer l'égalité entre deux points en montrant qu'ils satisfont chacun une même propriété caractéristique d'intersection.

L'intersection d'une corde et d'un cercle n'a pas été définie car elle peut se ramener à l'intersection d'un cercle et de son symétrique par rapport à cette corde.

5. Implémentation en Coq.

L'assistant de preuves Coq, basé sur le calcul des constructions, fait la distinction entre la sorte $Prop$ des propositions logiques dans laquelle la logique classique est acceptable et la sorte Set des objets construits.

5.1. Le point de départ.

L'idée initiale consistait à définir la géométrie par trois types mutuellement inductifs dans Set :
– les points, soit O, soit U soit le résultat d'une intersection,
– les droites par construction à la règle à partir de deux points,
– les cercles par construction au compas à partir de trois points.

Cependant cette construction demandait l'insertion de préconditions de type $Prop$, $A \neq B$ pour la droite (AB) ou pour le cercle de centre C et de rayon AB, (D) et (D') sécantes pour l'intersection de droites (en utilisant l'orientation), (C) et (C') sécants pour l'intersection de cercles (en utilisant les longueurs) et enfin (D) diamètre de (C) pour l'intersection d'une droite et d'un cercle (en utilisant la colinéarité). Il fallait donc

dans une même structure avoir des définitions dans Set et d'autres dans $Prop$ qui s'appelaient récursivement et mutuellement.

Une telle insertion n'est pas possible dans Coq.

5.2. Définitions, axiomes et constructions.

On pose l'existence des trois ensembles, l'ensemble des points du plan $Plan$, l'ensemble des distances $\mathbb{L}$ et l'ensemble des angles $\mathbb{A}$, des deux points O et U, de la propriété d'orientation $Clockwise$ et des fonctions $Distance$ et $Angle$. Les axiomes d'orientation et de métrique sont énoncés comme des propriétés (dans la sorte $Prop$).

Les droites, les cercles et les intersections sont des constructions, ce sont des types dépendants de figures de la sorte Set. Leurs définitions inductives font appel respectivement aux constructeurs $regle$, $compas$ et $intersection$ de droites sécantes, de cercles sécants et de cercle et diamètre. Leur code Coq a été donné aux paragraphes 4.1.1, 4.2.1. et 4.3.2. Enfin la fonction associant un point unique à chaque intersection est posée comme un axiome (cf §4.3.3.).

5.3. Correction de cette construction.

La présence d'axiomes dans une construction Coq lui fait perdre la garantie de cohérence.

Ce ne sont pas tant les axiomes sur les propriétés qui risquent d'introduire une incohérence, elles sont vraies dans la géométrie de Hilbert. On pourrait écrire la construction de Hilbert de la géométrie jusqu'à la preuve de l'existence de deux orientations possibles, puis démontrer tous les axiomes sur les propriétés de notre système, cette preuve en Coq de correction serait entièrement dans la sorte $Prop$.

Par contre, l'axiome associant un point construit unique à chaque intersection est beaucoup plus délicat dans la mesure où c'est lui qui permet de contourner la difficulté énoncé au paragraphe 5.1.1. Il ferme en quelque sorte la boucle de la construction mutuellement inductive des points, droites et cercles envisagée initialement. Il affirme l'existence constructive du point d'intersection, rappelons que l'ensemble des points est de type Set, dès lors que l'on peut prouver une propriété d'intersection de type $Prop$ entre deux figures. La question de la cohérence se focalise donc sur cet axiome, et elle est pour l'instant ouverte.

5.4. Complétude de la construction.

Pour vérifier que cet ensemble d'axiomes définit bien la géométrie euclidienne du plan à la règle et au compas, on démontre qu'il implique l'axiomatique de Hilbert, à l'exception, évidemment, de l'axiome de continuité de Cantor (ou celui de Dedekind). On obtient ainsi une garantie de complétude, la géométrie ainsi construite est celle des points constructibles du plan.

La preuve de l'implication des axiomes de Hilbert (à l'exception de celui de continuité) a entièrement été rédigée en utilisant l'assistant de preuves Coq. La plupart des axiomes de Hilbert s'obtiennent aisément, soit directement à partir des axiomes de propriétés, soit à partir de constructions très simples.

5.4.1. Les axiomes de Hilbert.

Ils sont classiquement divisés en cinq rubriques.

- Les axiomes d'incidence précisent les relations entre points et droites. Par exemple, l'existence de trois points non alignés se démontre en construisant le triangle équilatéral de base $[OU]$.
- Les axiomes d'ordre sont relatifs à la relation $entre$ sur trois points. Par exemple, étant donnés deux points distincts A et B, il existe un troisième point C tel que B est entre A et C. L'intersection du cercle de centre B et de rayon AB avec la droite (AB) construit un tel point. Autre exemple, la propriété de Pasch, elle est développée ci-dessous.

– Les axiomes de congruence définissent les rapports d'égalité entre les longueurs de segments ou mesures d'angles. Ces axiomes sont très proches de ceux énoncés au chapitre 3.
– Les axiomes de continuité sont au nombre de deux, le premier exprimant que la droite est archimédienne, le second que toute suite de segments emboîtés de longueurs tendant vers zéro converge sur un point (énoncé de Cantor). Seul le premier est démontré en utilisant l'axiome 3.8.
– Le postulat d'Euclide s'énonce : par un point donné, il ne passe qu'une parallèle à une droite donnée. Sa démonstration est développée ci-dessous.

5.4.2. La propriété de Pasch.

Elle s'énonce ainsi : étant donnés trois points non colinéaires A, B, C et une droite (D) ne passant par aucun de ces points, si (D) coupe le segment $[AB]$, elle coupe l'un des deux autres segments $[BC]$ ou $[CA]$.

La preuve s'établit par cas (en utilisant l'axiome 2.4) selon l'orientation des points de constructions de la droite (D) et des points A, B, C. Pour chaque cas, on utilise le fait qu'une droite ayant deux de ses points orientés différement par rapport aux points de construction d'une seconde droite coupe celle-ci.

5.4.3. Le postulat d'Euclide.

La construction de la parallèle à une droite passant par un point donné, lorsque celui-ci est extérieur à la droite se fait en construisant au compas le quatrième sommet du parallélogramme ayant pour sommets ce point et les points de construction de la droite.

La preuve que cette droite ne coupe pas l'autre n'est pas très longue, elle se fait par l'absurde en utilisant le fait que le centre du parallélogramme est centre de symétrie échangeant une droite en l'autre.

La réciproque est beaucoup plus longue (fig.3). On montre qu'une droite (D) distincte de la parallèle (CE) à la droite (AB) construite grâce au parallélogramme $BACE$ coupe la droite (AB). On montre tout d'abord l'existence d'un point I de (D) du même côté que B par rapport à (CE), puis on construit le point J de (BC) tel que (IJ) est parallèle à (CE). La distance étant archimédienne, il existe un entier $n \neq 0$ tel que l'extrémité K du segment $CK = n.CJ$ soit de l'autre côté de (AB) que C (au-delà de B). On construit le point L de (D) tel que $CL = n.CI$. Il reste à montrer que (KL) est parallèle à (IJ), cette démonstration utilisant des égalités d'angles et l'axiome 4.4, pour conclure que la droite (D) ayant deux points, un de chaque côté de (AB) est sécante à (AB).

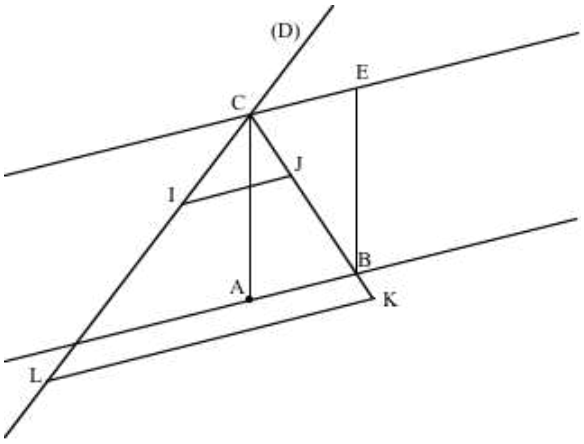


FIG. 3 – **Unicité de la parallèle à la droite (AB) menée par le point C.**

5.5. A propos de l'axiome de continuité.

L'axiome de continuité permet de parler de droite réelle (c'est-à-dire qu'il existe une bijection entre les points d'une droite et l'ensemble des nombres réels). L'existence de ces points de coordonnées irrationnelles transcendantes ou racines de polynômes de degré supérieur à 2 sont utiles en analyse (étude de courbes) ou en mécanique (roulement sans frottement). A ma connaissance, l'étude de la géométrie plane se concentre sur les points constructibles.

Dans notre construction, l'ensemble des points est posé au départ. Rien n'interdit d'imaginer qu'il contient tous les points de coordonnées réelles, puisque tous les axiomes sont vrais dans le plan réel. Toutefois, les seuls points sur lesquels on travaille, sont, à part les points O et U donnés au départ, les points que l'on peut construire par intersection de droites et de cercles. Cela est conforme à notre volonté de formaliser la géométrie plane euclidienne à la règle et au compas. Puisque d'une preuve, on veut extraire la constrution d'une figure à la règle et au compas, il n'est pas question de démontrer l'existence d'un polygone régulier de n côtés pour tout n.

6. Perspectives et développements.

6.1. Améliorations et simplifications.

Le système décrit ci-dessus est exactement celui utilisé dans la preuve de l'implication des axiomes de Hilbert. Sans modifier les principes qui ont guidé son élaboration, un certain nombre de simplifications et d'améliorations peuvent y être apportées. Il ne sera pas nécessaire de réécrire toute la preuve, il suffira de montrer que le système amélioré entraîne le système décrit ci-dessus.

Par exemple, il est clair qu'il n'est pas nécessaire de générer les deux intersections de deux cercles, l'intersection à droite de la droite des centres suffit. Pour obtenir l'intersection à gauche, il suffira de changer l'ordre des cercles dans le constructeur d'intersection de cercles. Il en va de même pour l'intersection d'un cercle et d'un diamètre.

Un autre exemple consiste à séparer l'axiome 4.1 (sur l'indépendance de l'angle par rapport aux points définissant les côtés) en deux axiomes, l'un exprimant la symétrie :

$$\forall\ A\ B\ C,\ \widehat{BAC} = \widehat{CAB}$$

et l'autre l'indépendance par rapport à un des points définissant un côté :

$$\forall\ A\ B\ C\ D,\ A \neq B \wedge D \in]AB) \Rightarrow \widehat{BAC} = \widehat{DAC}.$$

Cette solution diminuera le nombre de preuves du style $A \neq B$ dans les démonstrations.

6.2. Écriture d'un jeu de tactiques.

Coq offre la possibilité aux utilisateurs de définir des tactiques spécialisées sur des tâches spécifiques au domaine étudié. Nous allons utiliser cette facilité pour simplifier l'écriture des preuves de deux façons.

La première est l'automatisation de la preuve des petites hypothèses présentes dans nombre de théorèmes et dont la preuve est généralement omise car triviale dans une démonstration à la main. Ce sont des preuves portant sur la distinction entre deux points, sur l'alignement ou le non alignement de trois points, sur l'ordre de trois points alignés, sur des manipulations triviales d'égalité de longueurs ou d'angles. Ces tactiques ne devront pas être bloquantes, et laisser à l'utilisateur le soin de la preuve lorsque celle-ci ne pourra être directement déduite de l'environnement.

La seconde consiste à réunir en une seule tâche une construction (de point, de droite ou de cercle). Ces constructions se déduisent de théorèmes ayant une conclusion de type spécification d'un de ces objets. Par

exemple, sous les hypothèses $D1$ est une droite, $D2$ est une droite et $D1$ et $D2$ sont sécantes on sait construire un point M vérifiant les propriétés $M \in D1$, $M \in D2$ et $\forall\ N, N \in D1 \wedge N \in D2 \Rightarrow N = M$. La tactique cherchera à vérifier automatiquement les hypothèses, en cas d'échec, demandera à l'utilisateur de le faire, puis rajoutera dans l'environnement l'objet construit sous un nom donné en paramètre de la tactique et les propriétés vérifiées par cet objet.

6.3. Interface utilisateur.

Cette interface aura pour but de réaliser avec Coq en arrière plan, des démonstrations de géométrie plane dans un cadre aussi proche que possible de celui des démonstrations papier sans avoir besoin de connaître le langage et le fonctionnement de Coq. L'étude de cette interface s'effectue sur les quatre points suivants.

6.3.1. La fenêtre des figures.

Des logiciels de tracés interactifs de figures géométriques existent déjà tel que Geoview ou GeoGebra. Il faudrait en choisir un qui s'interconnecte avec l'environnement de la preuve en Coq. La séparation entre constructions (points, droites, cercles) et propriétés facilitera le travail de traduction des informations. Il faudra également choisir la meilleure option lorsque le logiciel de dessin ne parviendra pas à tracer une figure satisfaisant toutes les hypothèses.

6.3.2. La fenêtre des hypothèses et celle des conclusions.

Il faudra également traduire la liste des prémisses formant l'environnement et la liste des buts à démontrer en des phrases en langue française formant les hypothèses et les conclusions du problème en cours. Ce traducteur devra être facilement incrémenté pour prendre en compte de nouvelles définitions.

6.3.3. La fenêtre de la démonstration.

Dans l'autre sens, l'utilisateur devra pouvoir écrire sa démonstration en une suite de phrases en français qui seront automatiquement traduites en tactiques et commandes Coq. Pour simplifier le travail, on envisage de contraindre l'utilisateur à n'employer que des phrases types. Par exemple, la phrase : *Soit I l'intersection des droites (AB) et (D).* serait interprétée pour tenter d'appliquer la tactique d'intersection de deux droites. Une boîte de messages d'erreur permettra d'afficher les raisons des échecs. Un succès générera une nouvelle liste de buts avec des listes de prémisses pour chacun d'entre eux.

Dans une version plus élaborée, il pourrait être intéressant d'ajouter une fenêtre des lemmes utilisables sur le but courant. Cette fenêtre présenterait une traduction française de ces instances de lemmes.

6.3.4. L'enrichissement en définitions.

Enfin, dernière exigence à fixer au cahier des charges : le logiciel devra pouvoir accepter de nouvelles définitions, par exemple la bissectrice d'un angle. Si la définition est donnée en Coq, il n'y a pas de difficulté, par exemple, la bissectrice sera définie par sa construction à la règle et au compas. Par contre, si l'on veut que la définition soit faite en français, il faudra donner des phrases types permettant de faire des définitions tant de nouvelles propriétés (par exemple être un triangle rectangle) que de nouvelles constructions (comme la bissectrice) voire de nouvelles mesures (comme l'aire).

En effet ce logiciel ne se conçoit que comme un outil susceptible d'accueillir les contributions des utilisateurs dans tous les domaines : nouveaux théorèmes et leurs démonstrations, meilleures démonstrations de théorèmes déjà en place, nouvelles notions.

7. Conclusion.

Nous avons présenté une axiomatique de la géométrie euclidienne plane, dans laquelle une orientation et une métrique sont définies par des axiomes, puis les constructions des droites à la règle, des cercles au compas et des points par intersection de ces figures sont définies. Il est démontré à l'aide de l'assistant de preuves Coq que de ce système se déduisent tous les axiomes de Hilbert à l'exception de celui de continuité.

Ce système devrait être une bonne base pour la réalisation d'un outil d'aide à la démonstration en géométrie plane destiné aux professeurs et aux élèves des lycées. En effet, il permet de développer les raisonnements en s'appuyant sur la construction de figures.

Références

[BGP03] Y. BERTOT, F. GUILHOT AND L. POTTIER. *Visualizing Geometrical Statements with GeoView*. In User Interfaces for THeorem Provers (UITP'2003), vol. 103 of Electr. Notes Theor. Comput. Sci. pp. 49-65, 2003.

[Cab06] CABRILOG. *Cabri 3D : logiciel de géométrie interactive 3D*. `http:\\www.cabri.com, 2006`.

[Coq05] COQ DEVELOPMENT TEAM, INRIA AND LRI. *The Coq Proof Assistant Reference Manual, 2005*. `http:\\coq.inria.fr\doc\main.html`.

[DDS00] C. DEHLINGER, J.-F. DUFOURD AND P. SCHRECK. *Higher-Order Intuitionistic Formalization and Proofs in Hilbert's Elementary Geometry*. In Automated Deduction in Geometry 2000, vol. 2061 of LNCS, pp. 306-324, Springer-Verlag, 2000.

[Gui05] F. GUILHOT. *Formalisation en Coq et visualisation d'un cours de géométrie pour le lycée*. Revue des Sciences et Technologies de l'Information, TSI, vol. 24, Langages applicatifs, pp. 1113-1138, Lavoisier, 2005.

[Har02] R. HARTSHORNE. *Geometry : Euclid and beyond*. Ed. Springer, 2002.

[Hil71] D. HILBERT. *Les fondements de la géométrie*. Éd. P. Rossier, Dunod, 1971.

[Knu91] D. KNUTH. *Axioms and Hulls*. Number 606 in LNCS. Springer-Verlag, 1991.

[MF03] L. I. MEIKLE AND J. D. FLEURIOT. *Formalizing Hilbert's Grundlagen in Isabelle/Isar*. In TPHOLs'2003, vol. 2758 of LNCS, pp. 319-334, Springer-Verlag, 2003.

[Qua92] A. QUAIFE. *Automated Development of Fundamental Mathematical Theories*. Éd. Kluwer, 1992.

[vP95] J. VON PLATO. *The axioms of constructive geometry*. Annals of Pure and Applied Logic, 76, pp. 169-200, 1995.

Implémentations sûres de sessions typées

Cédric Fournet

Microsoft Research

Résumé

De nombreux programmes répartis peuvent s'écrire comme des échanges de messages entre participants, en fixant à l'avance la structure et l'ordonnancement des messages. Ces sessions (aussi appelées contrats, ou protocoles) simplifient la programmation : chacun des participants doit juste suivre son rôle dans la session, en supposant que les autres participants feront de même. Lorsque ces participants s'accordent une confiance relative, la bonne exécution globale de la session ne va pas de soi, et chacun doit aussi suivre et vérifier ce que font les autres. Confronté à ce problème de sécurité, le programmeur doit renoncer à la simplicité abstraite de la session et considérer les détails des protocoles de communications sous-jacents.

Nous développons des langages et des outils pour décrire des sessions et générer leurs implémentations réparties. Nous étendons ML par des sessions typées, telles qu'un programme bien typé suit son rôle dans toute session. Nous développons aussi un compilateur qui, pour un type de session donné, génère un protocole cryptographique sur mesure qui protège ces programmes bien typés : lors de l'exécution répartie d'une session, même en présence de participants qui trichent (en ne suivant pas nécessairement notre protocole et nos types), tout se passe comme si tous les participants suivaient bien la session.

Ce travail a été réalisé en collaboration avec Ricardo Corin, Pierre-Malo Dénielou, Karthikeyan Bhargavan, et James Leifer.

Gagner en passant à la corde

Jean-Christophe Filliâtre

CNRS
LRI, Université Paris Sud, 91405 Orsay, France
INRIA Futurs (ProVal), 91893 Orsay, France
filliatr@lri.fr

Résumé

Cet article présente une réalisation en Ocaml de la structure de cordes introduite par Boehm, Atkinson et Plass. Nous montrons notamment comment cette structure de données s'écrit naturellement comme un foncteur, transformant une structure de séquence en une autre structure de même interface. Cette fonctorisation a de nombreuses applications au-delà de l'article original. Nous en donnons plusieurs, dont un éditeur de texte dont les performances sur de très gros fichiers sont bien meilleures que celles des éditeurs les plus populaires.

1. Introduction

La dixième édition du concours de programmation de l'ICFP [4] a été l'occasion de découvrir ou de redécouvrir la structure de *cordes* (en anglais *ropes*). Il s'agit d'une structure de données pour les chaînes de caractères introduite par Boehm, Atkinson et Plass [3] dans le cadre du développement du langage Cedar à Xerox PARC. Dans cet article, les auteurs motivent l'introduction d'une structure de données alternative pour les chaînes de caractères par les arguments suivants :

- les chaînes doivent être *immuables* — autrement dit, elles doivent être réalisées par un type de données *persistant* [6] ;
- les opérations usuelles telles que la concaténation ou l'extraction d'une sous-chaîne doivent être *efficaces*, notamment en espace ;
- les chaînes ne doivent pas être limitées en taille, et les opérations doivent rester efficaces sur les très longues chaînes ;
- enfin, il doit être possible de considérer d'autres types de séquences de caractères, tels que des fichiers par exemple, comme des chaînes de caractères.

Il est clair que les chaînes de caractères fournies en standard dans tous les langages de programmation ne remplissent pas ces critères. Bien au contraire, elles peuvent être (et sont donc) modifiées en place, sont inefficaces quant à la concaténation et à l'extraction d'une sous-chaîne (car impliquant des copies) et enfin sont parfois sévèrement limitées en taille. Quant à la dernière fonctionnalité, elle n'est tout simplement jamais proposée.

En réponse à ces déficiences des chaînes de caractères usuelles, Boehm, Atkinson et Plass proposent la structure de *cordes*, où les chaînes sont représentées par des arbres binaires dont les nœuds représentent des concaténations et où les feuilles sont des chaînes de caractères usuelles. L'article décrit quelques algorithmes sur les cordes, dont un algorithme de rééquilibrage *a posteriori*, discute quelques points de réalisation dans les langages C et Cedar, et présente des tests de performances.

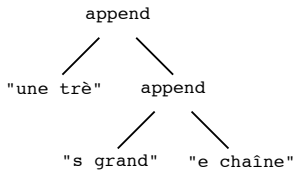
L'objectif du présent article est double. Tout d'abord, il s'agit de présenter une structure de données sans doute trop souvent négligée, ou simplement méconnue. En particulier, de nombreuses idées derrière la structure de cordes peuvent facilement être réutilisées dans d'autres contextes. Ensuite, nous allons au-delà de l'article original en réécrivant naturellement la structure de cordes comme un

foncteur. Celui-ci transforme une structure de séquence en une autre structure de même interface. Les applications de cette fonctorisation sont nombreuses et nous en décrivons plusieurs. En particulier, nous présentons la réalisation d'un mini éditeur de texte utilisant les cordes, dont les performances sont bien meilleures que celles de tous les éditeurs que nous avons testés sur des fichiers de très grande taille.

Cet article est organisé de la façon suivante. La section 2 décrit la structure de cordes et sa réalisation en Ocaml. La section 3 présente ensuite cette réalisation sous la forme d'un foncteur, dont les applications sont décrites dans la section 4. Enfin nous concluons avec des perspectives inspirées par cette réalisation des cordes. Cet article est illustré par du code écrit dans le langage Ocaml [1]. L'intégralité du code est disponible à l'adresse `http://www.lri.fr/~filliatr/software.fr.html`.

2. Réalisation des cordes en Ocaml

Cette section présente une réalisation des cordes en Ocaml. Sauf mention explicite du contraire, il s'agit d'idées présentes dans l'article original [3]. Comme nous l'avons expliqué dans l'introduction, une corde n'est rien d'autre qu'un arbre binaire dont les feuilles sont des chaînes (usuelles) de caractères, et dont les nœuds doivent être vus comme des concaténations. Ainsi, la corde



est l'une des multiples façons de représenter la chaîne `"une très grande chaîne"`. Une corde peut donc être directement codée par le type Ocaml suivant :

```
type rope = Str of string | App of rope × rope
```

Cependant, deux considérations nous poussent à raffiner légèrement ce type. D'une part, de nombreux algorithmes ont besoin d'un accès efficace à la longueur d'une corde, notamment pour décider de descendre dans le sous-arbre gauche ou dans le sous-arbre droit d'un nœud `App`. Il est donc souhaitable d'ajouter la taille de la corde comme une décoration de chaque nœud interne. D'autre part, il est important de pouvoir partager des sous-chaînes entre les cordes elles-mêmes et avec les chaînes usuelles qui ont été utilisées pour les construire. Dès lors, plutôt que d'utiliser un nœud `Str` pointant sur une chaîne Ocaml complète, on va préférer un nœud désignant une sous-chaîne de cette chaîne Ocaml. On obtient donc le type suivant :

```
type rope =
  | Str of string × int × int
  | App of rope × rope × int
```

Un nœud `Str(`s`,`o`,`n`)` représente la sous-chaîne `s`$[o..o+n-1]$ c'est-à-dire la portion de la chaîne s de longueur n située au caractère o. Un nœud `App(`r_1`,`r_2`,`n`)` représente la concaténation des deux cordes r_1 et r_2, dont la longueur totale est n. (On aurait pu tout aussi bien stocker les tailles de r_1 et r_2 dans le nœud ; mais ne garder que la taille totale est plus économe en taille mémoire, sans perte d'efficacité en pratique.)

Dans un premier temps, nous présentons successivement des opérations d'accès, de construction et de modification sur les cordes. Puis nous montrons comment les cordes peuvent être équipées de curseurs afin d'améliorer l'efficacité de certaines opérations.

2.1. Opérations d'accès

L'accès à la longueur d'une corde est immédiat, par construction :

```
let length = function Str (_,_,n) | App (_,_,n) → n
```

L'accès à son i-ième caractère nécessite de descendre dans l'arbre jusqu'à la bonne feuille. Nous supposons ici que les caractères sont indexés à partir de 0, comme les chaînes usuelles d'Ocaml. Nous supposons d'autre part que la vérification de la validité de l'accès a été effectuée en amont. La partie récursive de l'accès s'écrit alors ainsi :

```
let rec get i = function
  | Str (s, ofs, _) →
      s.[ofs + i]
  | App (t1, t2, _) →
      let n1 = length t1 in if i < n1 then get i t1 else get (i - n1) t2
```

On voit ici l'intérêt d'obtenir la taille de `t1` en temps constant. La complexité de l'accès est donc bornée par la hauteur de l'arbre. Comme nous l'indiquerons dans la section suivante les cordes peuvent être équilibrées, afin de garantir un accès au pire logarithmique en fonction du nombre de nœuds.

2.2. Opérations de construction

La corde vide peut être représentée par une chaîne de longueur 0 :

```
let empty = Str ("", 0, 0)
```

La construction d'une corde à partir d'une chaîne Ocaml est immédiate :

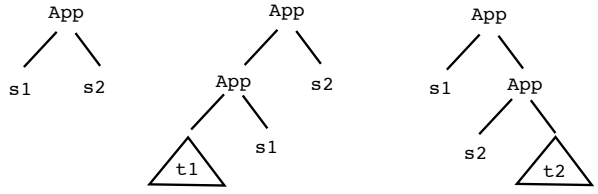
```
let of_string s = Str (s, 0, String.length s)
```

On notera cependant qu'afin de garantir le caractère immuable des cordes il faudrait en toute rigueur *copier* la chaîne `s` au lieu de simplement pointer vers celle-ci. Alors, et alors seulement, le caractère abstrait du type `rope` nous garantirait sa persistance.

Concaténation. La concaténation de deux cordes est *a priori* immédiate, le nœud `App` étant précisément là pour représenter une concaténation :

```
let mk_app t1 t2 = App (t1, t2, length t1 + length t2)
```

On note qu'il s'agit donc d'une opération en temps constant. Cependant, des concaténations massives vont amener le nombre de nœuds à croître rapidement, et donc la hauteur de l'arbre, au détriment des performances des autres opérations. Deux idées différentes permettent de maîtriser le nombre de nœuds et la hauteur de l'arbre. La première consiste à réaliser effectivement la concaténation des chaînes lorsque de petites feuilles se retrouvent côte à côte dans l'arbre. On peut choisir par exemple d'effectuer la concaténation des feuilles `s1` et `s2` dans les trois situations suivantes :



Ceci peut être effectué par l'opération de concaténation sur les cordes. Avec une longueur limite (arbitraire) de 256 caractères cela donne le code suivant :

```
let append t1 t2 = match t1, t2 with
  | Str (s1, ofs1, len1), Str (s2, ofs2, len2) when len1 <= 256 && len2 <= 256 →
      Str (String.sub s1 ofs1 len1 ^ String.sub s2 ofs2 len2, 0, len1 + len2)
  | App (t1, Str (s1, ofs1, len1), _), Str (s2, ofs2, len2) when ... →
      App (t1, ...<idem>...)
  | Str (s1, ofs1, len1), App (Str (s2, ofs2, len2), t2, _) when ... →
      App (...<idem>..., t2)
  | _ →
      mk_app t1 t2
```

Comme on peut le constater, cette opération est particulièrement efficace lorsque l'on construit la corde par concaténations successives de petites chaînes à l'un de ses bouts, ce que l'on peut considérer comme une opération relativement courante.

La seconde amélioration à apporter à la construction des cordes pour en limiter la hauteur consiste naturellement à effectuer un *rééquilibrage* de l'arbre. Bien que l'article original évoque la possibilité d'un équilibrage incrémental, en utilisant des AVL par exemple, il privilégie un rééquilibrage *a posteriori*, effectué soit sélectivement lorsque la hauteur de l'arbre devient trop importante, soit explicitement à la demande de l'utilisateur. L'article décrit une méthode de rééquilibrage originale, basée sur la longueur des cordes plutôt que sur leur nombre de nœuds. Mais une méthode directe consistant à transformer l'arbre en un arbre binaire (presque) complet à partir de l'ensemble de ses feuilles est tout aussi efficace. Le code en est donné dans l'appendice A.

Extraction de sous-corde. L'extraction d'une sous-corde consiste à ne conserver que les parties de la corde impliquées dans la portion concernée, en modifiant éventuellement les nœuds `Str` se trouvant à cheval sur celle-ci. Si la sous-corde à extraire s'étend du caractère `start` (inclus) au caractère `stop` (exclu), et en supposant une fois encore les vérifications de validité déjà effectuées en amont, l'extraction peut s'écrire de la manière suivante :

```
let rec sub start stop = function
  | Str (s, ofs, _) →
      Str (s, ofs+start, stop-start)
  | App (t1, t2, _) →
      let n1 = length t1 in
      if stop <= n1 then sub start stop t1
      else if start >= n1 then sub (start-n1) (stop-n1) t2
      else mk_app (sub start n1 t1) (sub 0 (stop-n1) t2)
```

Dans le cas du nœud `App`, on distingue trois situations, suivant que la sous-corde se trouve dans `t1` (premier cas), dans `t2` (deuxième cas) ou bien à cheval sur `t1` et `t2` (troisième cas). Dans ce dernier cas, on réutilise la fonction `mk_app` ci-dessus qui construit le nœud `App` en calculant la longueur totale (*smart constructor*).

Ce code peut encore être légèrement amélioré pour partager les sous-arbres qui se trouvent être identiques entre la corde initiale et la sous-corde extraite. Pour cela, il suffit de retourner directement l'argument de `sub` dès lors que la portion à extraire constitue l'intégralité de la corde :

```
let rec sub start stop t =
  if start = 0 && stop = length t then t else ...<même code que ci-dessus>...
```

2.3. Opérations de modification

Les opérations de « modification » des cordes doivent respecter le caractère immuable de celles-ci, et donc renvoyer de nouvelles cordes. Pour insérer des caractères dans une corde, ou en supprimer, on peut donc simplement utiliser les opérations de concaténation et de sous-chaîne déjà écrites. Si on note la concaténation par l'opérateur infixe ++, alors on peut définir la suppression du i-ième caractère d'une corde ainsi :

```
let delete t i = sub t 0 i ++ sub t (i + 1) (length t - i - 1)
```

De même, on peut définir l'insertion du caractère c juste après le i-ième caractère par

```
let insert t i c =
  sub t 0 i ++ of_string (String.make 1 c) ++ sub t i (length t - i)
```

on encore la modification du i-ième caractère en c par

```
let set t i c =
  sub t 0 i ++ of_string (String.make 1 c) ++ sub t (i + 1) (length t - i - 1)
```

Cependant, on note que ces fonctions descendent plusieurs fois dans l'arbre pour atteindre la même position. Il est donc préférable de les écrire de manière directe, avec un unique parcours de l'arbre. L'appendice B donne un tel code pour la fonction `delete`.

2.4. Curseurs

Il est fréquent de consulter ou de modifier une chaîne de manière répétée en un endroit précis, ou du moins sur un ensemble de positions proches les unes des autres. Dès lors, les descentes répétées vers le même endroit de la corde finissent par avoir un coût non négligeable par rapport à l'utilisation d'une chaîne de caractères usuelle. Pour y remédier, on peut ajouter aux cordes la notion de *curseur*, qui désigne une position dans la corde. Cette notion est présente dans l'article original et quelques détails de codage sont évoqués. Bien qu'il ne soit pas mentionné explicitement, on reconnaît là la description d'un *zipper* [5]. En effet, on peut facilement représenter un curseur par la donnée d'une feuille de la corde, d'un indice dans cette feuille et du contexte dans lequel apparaît cette feuille. Ce dernier élément peut être représenté simplement par le chemin menant de la feuille à la racine de la corde :

```
type path =
  | Top
  | Left  of path × rope
  | Right of rope × path
```

Le curseur est alors un enregistrement contenant l'ensemble des informations le définissant :

```
type cursor = {
  rpos: int;    (* position relative dans la feuille *)
  lofs: int;    (* position absolue de la feuille *)
  leaf: rope;   (* la feuille, de la forme Str (s,ofs,len) *)
  path: path;   (* le zipper *)
}
```

L'intérêt des curseurs est alors clair : des modifications locales au point désigné par le curseur peuvent maintenant être effectuées en temps constant, là où elles nécessitaient auparavant un temps proportionnel à la hauteur de la corde. Ainsi l'accès au caractère désigné par la curseur est immédiat :

```
let cursor_get c = let Str (s, ofs, len) = c.leaf in s.[ofs + c.rpos]
```

De même on peut écrire des fonctions d'insertion ou de suppression opérant sur un curseur en temps constant.

Passer de la corde au curseur et réciproquement se fait en utilisant les opérations usuelles d'ouverture et de fermeture du *zipper*. Ainsi la fonction `cursor_at` construit un curseur placé au caractère `i` de la corde `r`. Il s'agit de descendre jusqu'à la feuille concernée, tout en maintenant le contexte courant (argument `p` de `zip`) et la position courante dans la corde toute entière (argument `lofs`) :

```
let cursor_at r i =
  let rec zip lofs p = function
    | Str (_, _, len) as leaf  →
        { rpos = i - lofs; lofs = lofs; leaf = leaf; path = p }
    | App (t1, t2, _) →
        let n1 = length t1 in
        if i < lofs + n1 then
          zip lofs (Left (p, t2)) t1
        else
          zip (lofs + n1) (Right (t1, p)) t2
  in
  if i < 0 || i > length r then raise Out_of_bounds;
  zip 0 Top r
```

La fonction inverse reconstruit une corde à partir du contexte, en utilisant la fonction `mk_app` :

```
let rope_of_cursor c =
  let rec unzip t = function
    | Top → t
    | Left (p, tr) → unzip (mk_app t tr) p
    | Right (tl, p) → unzip (mk_app tl t) p
  in
  unzip c.leaf c.path
```

Le coût de ces deux opérations est proportionnel à la hauteur de l'arbre.

3. Fonctorisation

Jusqu'à présent, nous nous sommes contentés de coder en Ocaml les idées présentes dans l'article de Boehm, Atkinson et Plass [3]. Nous allons maintenant montrer comment la puissance du langage Ocaml peut être mise à profit pour aller plus loin. La première étape consiste à réécrire le code ci-dessus sous la forme d'un foncteur.

En effet, nous avons utilisé le type primitif de chaînes de caractères d'Ocaml pour les feuilles de nos cordes, mais cela n'avait rien d'une nécessité. Nous aurions pu tout aussi bien utiliser des tableaux ou des listes de caractères, et plus généralement encore des séquences de valeurs d'un autre type que celui des caractères. Plus généralement, il suffit de disposer d'une structure de données pour des *séquences* d'un certain *type de caractères*. On peut alors construire des cordes dont les feuilles seront réalisées par ces séquences-là. La structure de données obtenue a *la même signature* que celle de départ, à savoir celle de séquences pour le même type de caractères. Les cordes sont donc naturellement un foncteur transformant une structure de séquence en une autre structure de séquence de même interface.

Plus précisément, on peut se donner l'interface minimale suivante pour les structures de séquence :

```
module type STRING = sig
  type t
  type char
  val length : t → int
  val empty : t
  val singleton : char → t
  val get : t → int → char
  val append : t → t → t
  val sub : t → int → int → t
end
```

où `t` est le type des séquences et `char` le type de ses caractères. Les cordes sont alors un foncteur ayant le profil suivant :

```
module Rope(S : STRING) : STRING with type char = S.char
```

c'est-à-dire transformant un module `S` d'interface `STRING` en un module de même interface pour les mêmes caractères. Le codage de ce foncteur est immédiat, et en tout point semblable à ce que nous avons décrit dans la section précédente :

```
module Rope(S : STRING) = struct
  type t =
    | Str of S.t × int × int
    | App of t × t × int
  ...
end
```

En particulier, si on applique ce foncteur au module `String` de la bibliothèque standard d'Ocaml, on retrouve exactement le code décrit précédemment[1].

En pratique, le foncteur `Rope` fournit plus d'opérations en sortie qu'il n'en prend en entrée. On trouve en effet les opérations de modification dans le module résultat, alors qu'elles ne sont pas nécessaires dans le module d'entrée `S`. On note également que les deux opérations `append` et `sub` ne sont nécessaires dans `S` que si on souhaite effectuer l'optimisation décrite section 2.2 consistant à concaténer les petites feuilles.

4. Des avantages de la fonctorisation

Dans cette section nous présentons les avantages de la fonctorisation décrite dans la section précédente, au travers de plusieurs applications.

4.1. Itérations répétées du foncteur

Comme nous l'avons déjà dit, appliquer le foncteur `Rope` au module `String` nous redonne le codage des cordes présenté section 2, et le module obtenu a notamment une interface compatible avec celle de `String`. Dès lors, on peut appliquer de nouveau le foncteur `Rope`, pour obtenir un nouveau module de cordes. En répétant le processus, on obtient une infinité de structures de cordes :

```
module R1 = Rope(String)
```

[1] Le module `String` de la bibliothèque standard d'Ocaml ne définit pas de type `char` et il faut donc enrober `String` dans un module ajoutant `type char = Char.t`.

```
module R2 = Rope(R1)
module R3 = Rope(R2)
...
```

En quoi ces différentes structures de données diffèrent-elles ? Nous connaissons déjà `R1`. Les cordes de `R2` sont des arbres dont les feuilles sont des (sous-)cordes de `R1`, donc des arbres possédant un nœud « sous-corde » entre leur racine et leurs vraies feuilles, c'est-à-dire les chaînes Ocaml. De même, les cordes de `R3` sont des arbres dont les feuilles sont des cordes de `R2`, donc des arbres possédant deux nœuds « sous-corde » entre leur racine et leurs vraies feuilles, etc. Selon la stratégie adoptée concernant la concaténation des petites feuilles, on a donc des suspensions de calcul de sous-cordes au milieu des arbres, dont le nombre maximal est déterminé par le nombre d'applications successives du foncteur `Rope`. Le code disponible en ligne propose d'ailleurs un foncteur prenant un second argument pour contrôler les stratégies de concaténation des feuilles et de rééquilibrage.

Il est naturel de considérer le point-fixe obtenu en appliquant une infinité de fois le foncteur. Le type Ocaml résultant est le suivant :

```
type t =
  | Str of string
  | Sub of t × int × int
  | App of t × t × int
```

Sans surprise, c'est le type d'un arbre où l'on peut suspendre aussi bien des opérations de concaténation que des calculs de sous-cordes. Tout dépend ensuite de la stratégie choisie pour effectuer tout de suite, ou bien au contraire suspendre, les opérations de concaténation et de sous-corde. (Nous n'avons pas mené d'expérience pour comparer de telles stratégies.)

4.2. Tableaux

Dans le foncteur `Rope` le type des caractères n'est pas fixé, et il n'y a donc pas lieu de se limiter au type primitif `char`. Les cordes constituent donc autant une structure de tableaux que de chaînes de caractères. Pour obtenir des tableaux dont les éléments sont de type `elt` il suffit d'instancier le foncteur `Rope` avec le paramètre suivant :

```
module S = struct
  type char = elt
  type t = elt array
  let length = Array.length
  let empty = [||]
  let singleton l = [|l|]
  let get = Array.unsafe_get
  let append = Array.append
  let sub = Array.sub
end
```

Bien entendu, les tableaux obtenus n'ont pas l'efficacité des tableaux primitifs en ce qui concerne l'accès et la mise à jour. En revanche, ils offrent tous les avantages des cordes : persistance, opérations `append` et `sub` efficaces, et virtuellement aucune limite de taille. En particulier, ces cordes-tableaux permettent aisément de réaliser des tableaux redimensionnables, un problème récurrent avec Ocaml, où la limite de taille des tableaux est particulièrement basse sur les machines 32 bits (à savoir 4194303). Nous utiliserons de tels tableaux dans la section 4.4.

4.3. Fonctions vues comme des chaînes

L'article original introduisant les cordes insiste sur la possibilité pour une corde d'avoir des feuilles réalisées non pas par des chaînes usuelles mais par des *fonctions*. L'idée est de pouvoir représenter par exemple le contenu d'un très gros fichier, sans avoir à le charger en mémoire. La structure de cordes fournie dans la bibliothèque STL [2] reprend cette fonctionnalité.

Si nous n'avons pas inclus cette possibilité dans notre codage initial, c'est parce qu'il est aisé de l'obtenir *a posteriori* grâce à notre foncteur. En effet, puisque nous pouvons fixer le type de feuilles, nous pouvons choisir un type offrant l'alternative entre chaînes de caractères et fonctions :

```
module SorF : STRING = struct
  type char = Char.t
  type t = S of string | F of int × (int → char)
```

Une feuille de la forme `F(`n`,`f`)` représente une chaîne de longueur n dont le i-ième caractère est $(f\ i)$. Il est alors facile de réaliser les différentes opérations exigées sur ce type somme :

```
  let length = function S s → String.length s | F (n,_) → n

  let get t i = match t with S s → s.[i] | F (_,f) → f i

  let sub t ofs len = match t with
    | S s → S (String.sub s ofs len)
    | F (n, f) → F (len, fun i → f (i - ofs))
  ...
end
```

4.4. Un éditeur de texte

Pour terminer, nous présentons une application réaliste de la structure de cordes, à savoir un éditeur de texte. Si un éditeur de texte jouet peut être réalisé directement en utilisant les chaînes de caractères fournies par le langage, un éditeur réaliste se doit d'utiliser une structure adaptée aux fichiers de grande taille. Il est d'ailleurs frappant de constater que, malgré leurs efforts dans ce sens, les éditeurs populaires tels que Emacs ou vi montrent rapidement leurs limites sur un fichier contenant une unique ligne de plus de 7 millions de caractères (tel que le code ADN constituant le point de départ du dernier concours de programmation de l'ICFP [4]), que ce soit pour l'insertion, la suppression ou la recherche.

L'application de la structure de cordes aux éditeurs de texte est déjà mentionnée dans l'article de Boehm, Atkinson et Plass. Mais là où les auteurs suggèrent d'utiliser une corde pour représenter l'intégralité du texte en cours d'édition, nous nous proposons d'utiliser notre foncteur pour encore plus de flexibilité. En effet, une unique corde rend fastidieuse la gestion des différentes lignes du texte (il faut rechercher les sauts de ligne, ou maintenir leur position dans une table correctement synchronisée). Au lieu de cela, nous pouvons utiliser une corde pour représenter l'ensemble des lignes du texte et représenter chaque ligne, c'est-à-dire chaque élément de cette corde, par une corde dont les éléments sont des caractères. Nous commençons donc par construire une structure de corde pour les lignes :

```
module Line = Rope(String)
```

puis une autre structure de corde pour le texte, comme un tableau de lignes à la manière de la section 4.2 :

```
module Text = Rope(struct
  type char = Line.t
  type t = Line.t array
  let empty = [||]
  ...
end)
```

Le texte manipulé par l'éditeur peut alors être stocké dans une unique référence contenant une corde de type `Text.t` :

```
let text = ref Text.empty
```

Pour insérer le caractère `c` à la position `ofs` dans la ligne `l` du texte, il suffit de récupérer cette ligne avec `Text.get`, de la modifier avec `Line.insert` et enfin de mettre à jour la ligne avec `Text.set` :

```
let insert_char l ofs c =
  let line = Text.get !text l in
  let line' = Line.insert line ofs c in
  text := Text.set !text l line'
```

De même pour supprimer un caractère :

```
let delete_char l ofs =
  let line = Text.get !text l ln in
  let line' = Line.delete line ofs in
  text := Text.set !text l line'
```

Enfin, pour insérer un retour-chariot à la position `ofs` dans la ligne `l` du texte, il suffit de couper la ligne `l` à la position `ofs` avec deux appels à `Line.sub`, puis d'insérer le préfixe à la ligne `l` avec `Text.insert` et de positionner le suffixe en ligne `l+1` avec `Text.set` :

```
let insert_newline l ofs =
  let line = Text.get !text l in
  let prefix = Line.sub line 0 ofs in
  let suffix = Line.sub line ofs (Line.length line - ofs) in
  let r = Text.insert !text l prefix in
  text := Text.set r (l + 1) suffix
```

On note qu'il n'est pas nécessaire ici de « décaler » toutes les lignes : on utilise l'insertion dans la corde des lignes, tout comme on a utilisé l'insertion dans les cordes de caractères pour `insert_char`.

Sur la base de ce code, nous avons codé un mini éditeur de texte, supportant les fonctionnalités suivantes : insertion de texte, suppression, déplacements élémentaires (flèches, début et fin de ligne, page suivante, page précédente) et recherche. La partie graphique utilise la bibliothèque SDL. Le reste du code suit le patron du *modèle-vue-contrôleur* [7] : le modèle est le contenu du fichier, dont la gestion est présentée ci-dessus ; la vue assure la correspondance entre le modèle et la portion qui en est affichée, ainsi que la position du curseur graphique ; enfin le contrôleur est la boucle d'interaction gérant les événements clavier. Les lignes de code sont ainsi partagées :

module	**lignes**
graphisme	68
modèle	55
vue	54
contrôleur	86
total	**263**

Les résultats sont conformes aux attentes les plus optimistes : l'édition est absolument fluide quelle que soit la taille du fichier et quelle que soit la taille des lignes (nous avons fait des tests sur des fichiers de plus de 200 Mo et sur des lignes de plus de 8 Mo). Le code de ce mini-éditeur n'utilise même pas les curseurs présentés section 2.4, pour plus de simplicité. On peut donc espérer d'encore meilleurs résultats pour un éditeur utilisant un curseur (au sens des cordes) pour représenter la position actuelle du curseur graphique, et donc une insertion plus efficace. Enfin, il est important de noter que la persistance de la structure de corde est un avantage supplémentaire dans un logiciel tel qu'un éditeur de texte, car permettant une réalisation quasi-triviale du retour en arrière (*undo*).

5. Conclusion

Dans cet article, nous avons cherché à populariser une structure de données injustement méconnue, à savoir les *cordes*. Suivant le proverbe qui dit qu'il faut avoir plusieurs cordes à son arc, nous avons écrit cette structure de données comme un foncteur paramétré par la structure de séquences sous-jacente. Cette fonctorisation ouvre de multiples perspectives à l'utilisation des cordes, bien au-delà de ce qui est suggéré dans l'article original les introduisant. Nous avons notamment montré comment cette généricité permet de jeter facilement les bases d'un éditeur de texte à même d'appréhender des fichiers de très grande taille.

La réalisation de cette structure de cordes a permis de soulever au moins deux questions méritant plus ample investigation. D'une part, le type obtenu par point fixe section 4.1 constitue une variante des cordes où l'opération de sous-corde peut être suspendue au même titre que l'est celle de concaténation. Pour en tirer tous les avantages, il faudrait étudier les conditions sous lesquelles il est intéressant d'effectuer cette suspension, ou au contraire de ne pas l'effectuer. Il est clair par exemple que lors de plusieurs extractions successives de sous-cordes, les premières ont intérêt à être suspendues.

La seconde interrogation qui vient naturellement à l'esprit consiste à se demander si l'idée de suspendre des opérations en les représentant sous forme d'un arbre peut être généralisée avec profit à d'autres structures de données. On peut ainsi imaginer une structure d'ensembles où une opération telle que l'union, par exemple, n'est pas systématiquement effectuée. Restent à déterminer les conditions sous lesquelles une telle suspension peut être asymptotiquement avantageuse.

Références

[1] Le langage Objective Caml. `http://caml.inria.fr/`.

[2] SGI's extension to the C++ Standard Template Library : Ropes. `http://www.sgi.com/tech/stl/Rope.html`.

[3] Hans-Juergen Boehm, Russell R. Atkinson, and Michael F. Plass. Ropes : An alternative to strings. *Software - Practice and Experience*, 25(12) :1315–1330, 1995.

[4] Eelco Dolstra, Jur Hage, Bastiaan Heeren, Stefan Holdermans, Johan Jeuring, Andres Löh, Arie Middelkoop, Alexey Rodriguez, John van Schie, and Clara Löh. Morph Endo! Report on the Tenth Interstellar Contest on Fuun Programming. Technical Report UU-CS-2007-029, Institute of Information and Computing Sciences, Utrecht University, 2007.

[5] Gérard Huet. The Zipper. *Journal of Functional Programming*, 7(5) :549–554, Septembre 1997.

[6] Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998.

[7] Julien Signoles. Une approche fonctionnelle du modèle vue-contrôleur. In *Journées Francophones des Langages Applicatifs*, March 2005.

A. Rééquilibrage des cordes

Nous donnons ici un code simple pour rééquilibrer une corde *a posteriori*. Le code commence par construire la liste des feuilles (fonction `to_list`), tout en calculant le nombre de feuilles. Ensuite la fonction `build` reconstruit un arbre binaire équilibré, en coupant la liste en deux, récursivement.

```
let balance t =
  let rec to_list ((n, l) as acc) = function
    | Str _ as x → n + 1, x :: l
    | App (t1, t2, _) → to_list (to_list acc t2) t1
  in
  let rec build n l =
    assert (n >= 1);
    if n = 1 then begin
      match l with
        | [] → assert false
        | x :: r → x, r
    end else
      let n' = n / 2 in
      let t1, l = build n' l in
      let t2, l = build (n - n') l in
      mk_app t1 t2, l
  in
  let n, l = to_list (0, []) t in
  let t, l = build n l in
  assert (l = []);
  t
```

B. Suppression du i-ième caractère

La suppression d'un caractère dans une corde peut être optimisée en tenant compte des cas particuliers où ce caractère se trouve être au bord d'une feuille (à gauche ou à droite). La fonction `delete_rec` ci-dessous traite ces cas particuliers (trois premiers motifs), puis le cas général d'une feuille qu'il faut décomposer en deux nouvelles feuilles (où on notera le partage de la chaîne `s`) et enfin le cas d'un nœud de concaténation. Cette fonction suppose que la validité de l'indice `i` a été vérifiée en amont.

```
let rec delete_rec i = function
  | Str (_, _, 1) →
      assert (i = 0); empty
  | Str (s, ofs, len) when i = 0 →
      Str (s, ofs+1, len-1)
  | Str (s, ofs, len) when i = len-1 →
      Str (s, ofs, len-1)
  | Str (s, ofs, len) →
      mk_app (Str (s, ofs, i)) (Str (s, ofs+i+1, len-i-1))
  | App (t1, t2, _) →
      let n1 = length t1 in
      if i < n1 then
        mk_app (delete_rec i t1) t2
      else
```

```
      mk_app t1 (delete_rec (i-n1) t2)
```

On peut alors écrire la fonction `delete` qui effectue la vérification de validité avant d'appeler `delete_rec` :

```
let delete t i =
  if i < 0 || i >= length t then raise Out_of_bounds;
  delete_rec i t
```

Métaprogrammation fonctionnelle appliquée à la génération d'un DSL dédié à la programmation parallèle

J. Sérot[1] & J. Falcou[2]

1: LASMEA - UMR 6602 CNRS/UBP,
Campus des Cézeaux, 63177 Aubière CEDEX
Jocelyn.Serot@lasmea.univ-bpclermont.fr
2: IEF - U. Paris-Sud Orsay
joel.falcou@ief.u-psud.fr

Résumé

On décrit l'implémentation en MetaOcaml d'un petit langage dédié (DSL) à la programmation parallèle. Le langage repose sur la notion de *squelettes* qui autorisent la spécification de programmes parallèles par simple composition de constructeurs de haut niveau encapsulant des schémas communs et récurrents de parallélisme. On montre comment les facilités de *métaprogrammation* offertes par MetaOcaml permettent d'éliminer presque totalement le surcoût à l'exécution du code généré par rapport à une implantation du même programme écrite avec des primitives de bas niveau comme celles de mpi. Pour cela, la spécification haut niveau du programme est d'abord transformée en une représentation équivalente sous la forme d'un réseau de processus séquentiels communiquants puis cette représentation est utilisée pour générer *dynamiquement* le code exécuté par chaque processeur de la machine.

1. Introduction

De nos jours, le principal style de programmation utilisé pour exploiter les *clusters* de calcul est celui dit par *passage de messages*, supporté par exemple par des bibliothèques comme mpi. Le programmeur est amené à décomposer explicitement son application en processus communiquants et à ordonnancer "à la main" les communications entre ces processus. Cette approche, si elle permet à un programmeur expérimenté de tirer le meilleur parti d'une architecture donnée, conduit toutefois à des programmes longs à écrire et à mettre au point compte-tenu du très faible niveau d'abstraction utilisé.

Les *squelettes de parallélisation* ont été proposés en réponse à ce problème. Un *squelette* encapsule un *schéma de parallélisation* récurrent pour lequel une ou plusieurs implémentations efficaces sont connues. De tels squelettes se présentent classiquement comme des constructeurs génériques qui doivent être instanciés avec les fonctions spécifiques de l'application. Des exemples classiques de squelettes sont pipeline, farm et divide-and-conquer. Avec cette approche, le travail du programmeur se limite à choisir et combiner un ensemble de squelettes pris dans une base prédéfinie, sans qu'il ait à se préoccuper des détails de mise en œuvre du parallélisme sur l'architecture.

Les squelettes constituent un exemple de langage spécifique d'un domaine d'application (DSL, *Domain Specific Language*) répondant à un besoin d'abstraction. La nécessité de disposer d'un formalisme permettant de spécifier, puis de raisonner sur les programmes est une des raisons expliquant l'intérêt pour ces approches. De nombreux travaux ont été consacrés à la définition à l'implantation

de systèmes de programmation parallèles fondés sur les squelettes [1, 7, 2, 10, 9, 19]. Ces réalisations iffèrent essentiellement sur la manière dont le DSL correspondant est implanté. En pratique cela suppose de définir la syntaxe et la sémantique de ce DSL d'une part et les règles de transformation permettant de passer de cette spécification haut niveau à une implémentation concrète bas niveau – faisant appel à des primitives MPI par exemple – d'autre part.

Pour la plupart des réalisations citées, le DSL est implanté sous la forme d'une bibliothèque dédiée au sein d'un langage de programmation hôte (C++ ou Caml par exemple). Cette approche offre en effet deux avantages : elle évite d'avoir à implanter un analyseur lexical et syntaxique dédié et elle facilite l'interfaçage aux fonctions séquentielles (soit parce que ces fonctions sont écrites dans le langage hôte, soit parce que cet interfacage peut se faire via les facilités offertes par ce langage hôte). Mais, en contrepartie, cette manière de procéder conduit à des implémentations relativement peu efficaces, comparées au code "bas-niveau" qu'un programmeur aurait écrit pour le même problème. En effet, les squelettes étant par essence des objets d'ordre supérieur, leur support au sein d'un langage de programmation implique toujours un surcoût à l'exécution par rapport à un code bas-niveau faisant appel directement à des primitives MPI.

Nous expliquons dans ce papier comment les techniques d'*évaluation partielle* et de *métaprogrammation* – utilisés par ailleurs avec succès dans d'autres domaines d'applications – peuvent être exploitées pour résoudre le problème sus-cité. Plus précisément, nous décrivons l'implantation, en MetaOcaml, d'un petit DSL permettant

- la spécification de programmes parallèles sous la forme de combinaisons de squelettes
- la génération automatique d'un programme bas-niveau équivalent, sous la forme d'un ensemble de processus séquentiels communiquants
- l'exécution de ce programme sur une architecture de type *cluster* via la bibliothèque MPI.

La plan suivi est le suivant. La section 2 rappelle brièvement le principe de la métaprogrammation et comment celle-ci est supportée au sein du langage MetaOcaml. Dans la section 3 on explique en quoi cette technique présente un intérêt dans le contexte de la programmation parallèle par passage de messages. La section 4 est consacrée à la présentation d'un DSL dédié à programmation parallèle par squelettes. La sémantique de ce langage est explicitée en termes de réseaux de processus séquentiels communiquants puis on en dérive une implémentation en MetaOcaml. La section 5 présente les résultats expérimentaux obtenus avec cette implémentation. Un bref état de l'art est fait dans la section 6. La conclusion permet de faire le bilan des apports de ce premier prototype et d'indiquer les pistes de travail à court et moyen termes.

2. Métaprogrammation en MetaOcaml

De manière très générale, la métaprogrammation est la technique par laquelle un programme particulier peut analyser, transformer et générer d'autres programmes. La programmation *multi-niveaux* (*multi-staged programming*), supportée par le langage MetaOcaml, est une une instance particulière de cette technique.

MetaOcaml [20] est une extension du langage Ocaml [17] permettant

- de distinguer, grâce à un système du "quotation", au moins deux niveaux dans un programme : celui du code générant (le métaprogramme) et celui du code généré (les programmes objets).
- de générer et d'exécuter le code correspondant à ces programmes objets à l'exécution
- de garantir statiquement, via un système de typage adapté, que ceci se fait sans risque d'erreur à l'exécution

En pratique, cela signifie qu'un programme MetaOcaml peut, lors de son exécution, construire d'autres programmes et les exécuter. Ceci se fait à l'aide de trois constructeurs, nommés *Brackets*, *Escape* et *Run* :

- la construction `.< >.` (**brackets**) permet de délimiter les fragments des programmes objets et

donc à retarder l'exécution du code correspondant :

```
# let a = .< 1+2 >.;;
val a : int code = .<(1+2)>.
```

L'expression `1+2` n'est pas évaluée mais sa représentation est mémorisée dans la valeur `a`. Le type de `a` reflète ceci[1] : `int code` est le type des programmes qui, exécutés, produisent des valeurs de type `int`. Le fragment de programme correspondant peut être inséré dans d'autre programmes ou compilé et exécuté.

– l'opérateur `.~` (**escape**) permet justement d'insérer un fragment de programme objet dans un autre :

```
#let b = .< 3 * .~a >.;;
val b : int code = .<(3*(1+2))>.
```

– l'opérateur `.!` (**run**) compile et exécute un programme objet :

```
#let c = .!b;;
val c : int = 9
```

3. Métaprogrammation appliquée à la génération de code parallèle

Afin d'illustrer en quoi la métaprogrammation « à la MetaOcaml » présente un intérêt dans le cadre de la programmation parallèle considérons un programme (hypothétique) dans lequel un ensemble de $2N$ processus s'échangent des données selon le schéma suivant (cf figure 1) :

– un processus de rang pair envoie une donnée, générée par une fonction `f`, au processus de rang impair immédiatement supérieur puis attend une réponse de ce même processus
– un processus de rang impair reçoit une donnée du processus de rang pair immédiatement inférieur, applique une fonction `g` à cette donnée puis renvoie le résultat à ce même processus.

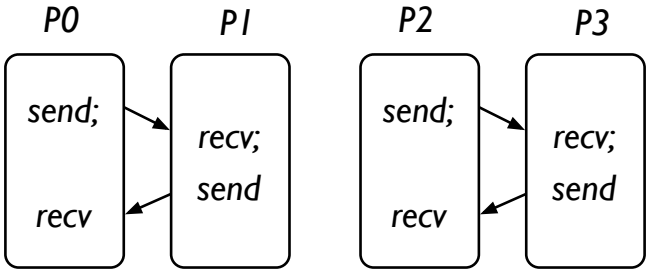


Fig. 1 – Exemple 1

Sur la quasi-totalité des architectures de type *cluster*, le modèle d'exécution est de type SPMD (*Single Program Multiple Data*) ce qui signifie que le programmeur écrit un seul programme qui s'exécute sur l'ensemble des processeurs. Ecrit avec des primitives de communication point à point de bas niveau de type `send` et `recv`[2] le code du programme précédent ressemble alors à ceci :

[1] Le type de `a` retourné par le compilateur MetaOcaml est en fait `('a, int) code` ; le paramètre `'a` est utilisé par le système de type pour garantir la correction du code dans le cas où des variables sont utilisées à plusieurs niveaux. Par souci de lisibilité, on l'omettra ici.

[2] On supposera par exemple que la signature de ces primitives est `val send : 'a -> pid -> unit` et `val recv : pid -> 'a`. De telles primitives sont par exemple offertes par la bibliothèque OcamlMpi.

```
let myrank = get_proc_rank() in
if myrank % 2 = 0 then
  send (f()) (myrank+1);
  recv (myrank+1)
else
  let y = recv (myrank-1)
  send (g y) (myrank-1);
```

Ici la primitive `get_proc_rank()` permet d'obtenir, à l'exécution, le rang[3] effectif du processeur qui exécute le programme et la conditionnelle décide, en fonction de ce rang, de la séquence d'instructions à exécuter.

On peut noter que dans ce programme, le schéma de communication est fixe et donc que, pour chaque processeur, le code à exécuter est connu dès que le rang du dit processeur est connu. On peut tirer parti de cette propriété pour réécrire le programme précédent de la manière suivante en MetaOcaml :

```
let pgm rank =
  if rank % 2 = 0 then
    .< send (f()) (rank+1); recv (rank+1) >.
  else
    .< let y = recv (rank-1) in send (g y) (rank-1) >.

let myrank = get_proc_rank() in
.!(pgm myrank)
```

Le code est exécuter est désormais *généré dynamiquement* au niveau de chaque processeur, puis exécuté. Cette technique est extrêmement puissante puisqu'elle permet de spécialiser – et donc d'optimiser – le code exécuté par chaque processeur en fonction de paramètres qui ne sont connus qu'à l'exécution (à la différence de systèmes de métaprogrammation opérant à la compilation comme les templates de C++ ou Template Haskell [12]). On peut objecter que ceci se fait au détriment du temps d'exécution mais dans la mesure où la génération du code n'est faite qu'une fois, au début de l'exécution du programme, le surcoût introduit peut être totalement recouvert par le gain en performance induit par le code optimisé.

Dans la suite, on va exploiter cette idée pour donner une implémentation efficace d'un DSL adapté à la programmation parallèle par squelettes.

4. Un DSL pour la programmation parallèle par squelettes

Le langage visé – appelons-le Skl– permet la spécification de programmes parallèles au moyen d'un nombre limité de combinateurs (les squelettes). Ces squelettes encapsulent complètement les activités de communication et de coordination associées au parallélisme. Les parties séquentielles de l'application sont spécifiées sous la forme de fonctions passées en paramètres. La structure de l'application est statique (pas de création / suppression dynamique de processus).

On décrit ici un jeu limité de squelettes, qui nous a servi à valider l'approche[4] :

[3] On utilise ici la terminologie mpi. Le rang permet de distinguer à l'exécution les copies d'un même programme dans un modèle SPMD.

[4] L'extension de ce jeu ne pose pas de problème *a priori* dès lors que ces squelettes peuvent être décrits dans les termes de l'algèbre de processus détaillée par la suite.

– le squelette PIPE encapsule un parallélisme de type flux : n étapes de calcul sont enchaînées sur les données d'entrée, chaque étape pouvant s'exécuter en parallèle sur des données distinctes ;
– le squelette PARDO encapsule un parallélisme de type tâche : n opérations sont appliquées en parallèle sur chaque donnée d'entrée ;
– le squelette FARM encapsule un parallélisme de type données : une même opération est appliquée à l'ensemble des données d'entrées ; le modèle d'exécution sous-jacent est celui dit en "ferme de processus", où un processus "maître" distribue dynamiquement les données à traiter à un ensemble de processus esclaves et collecte les résultats, afin de réaliser un équilibre de charge.

Dans la description précédente, les "étapes" (resp. "opérations") sont elles-mêmes des squelettes ; en d'autres termes, le langage supporte naturellement l'*imbrication* des squelettes. L'insertion des fonctions séquentielles se fait donc via un "squelette" particulier, nommé Seq.

La syntaxe abstraite des programmes SKL est donc la suivante :

$$\begin{array}{lcl} \Sigma & ::= & \mathsf{Seq}\ f \\ & | & \mathsf{Pipe}\ \Sigma_1 \ldots \Sigma_n \\ & | & \mathsf{Pardo}\ \Sigma_1\ \ldots \Sigma_n \\ & | & \mathsf{Farm}\ n\ \Sigma \\ f & ::= & \textit{fonction de calcul séquentielle} \\ n & ::= & \textit{entier} \geq 1 \end{array}$$

Imaginons par exemple un programme pouvant se décomposer en trois étapes : une étape de production de données (réalisée par une fonction f), une étape de traitement de ces données en parallèle à l'aide d'une ferme à trois esclaves (chacun exécutant la même fonction g) et une étape de traitement des résultats (réalisée par une fonction h) (cf figure 2). Ce programme s'écrit de la manière suivante en SKL :

$$\sigma = \mathsf{Pipe}(\mathsf{Seq}\ f, \mathsf{Farm}(3, \mathsf{Seq}\ g), \mathsf{Seq}\ h)$$

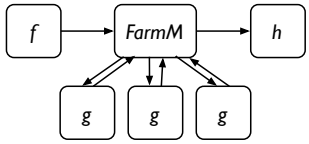


FIG. 2 – Exemple 2

4.1. Sémantique

On donne ici une sémantique des programmes SKL en termes de réseaux de processus séquentiels communiquants. Cette sémantique servira de base à la fonction d'interprétation qui, codée en METAOCAML, va permettre de générer le code à exécuter par chaque processeur de la machine parallèle cible.

On commence par définir ce que l'on entend exactement par réseau de processus séquentiels communiquant (RPSC).

Formellement, un RPSC est un triplet $\pi = \langle P, I, O \rangle$ où
– P est un ensemble de processus étiquetés, c-à-d. de paires (pid, σ) où pid est un identificateur (unique) de processus et σ un triplet contenant : une liste de *prédécesseurs* (pid des processus

p pour lesquels il existe un canal de communication de p au processus en question), une liste de *successeurs* (*pid* des processus p pour lesquels il existe un canal de communication du processus en question à p) et un descripteur Δ. On note $\mathcal{L}(\pi)$ l'ensemble des *pid*s d'un réseau de processus π. Pour un processus p, ses prédécesseurs, successeurs and descripteur seront notés $\mathcal{I}(p)$, $\mathcal{O}(p)$ et $\delta(p)$ respectivement.

- $I(\pi) \subseteq \mathcal{L}(\pi)$ désigne l'ensemble des processus p pour lesquels $\mathcal{I}(p) = \emptyset$
- $O(\pi) \subseteq \mathcal{L}(\pi)$ désigne l'ensemble des processus p pour lesquels $\mathcal{O}(p) = \emptyset$

Le descripteur de processus Δ est une paire $(instrs, kind)$ où $instrs$ est une séquence de *macro-instructions* et $kind$ un drapeau (la signification de ce drapeau sera donnée plus loin).

$$\begin{array}{ll} \Delta & ::= \langle instrs, kind \rangle \\ instrs & ::= instr_1, \ldots, instr_n \\ kind & ::= \mathsf{Regular} \mid \mathsf{FarmM} \end{array}$$

La séquence de macro-instructions décrivant le comportement du processus est implicitement itérée (en première approximation on considère que les processus ne se terminent pas ; gérer de manière "propre" la terminaison des processus complique la description du modèle et nous avons choisi de ne pas détailler ce point ici).

Les macros-instructions utilisent toutes un mode d'adressage *implicite* se référant à quatre variables nommées iv, ov, q et iws. Le jeu de macro-instructions est décrit ci-dessous. Dans les explications afférentes, p désigne le processus exécutant l'instruction en question.

$$\begin{array}{lll} instr & ::= & \mathsf{SendTo} \mid \mathsf{RecvFrom} \mid \mathsf{Comp}\ \text{fid} \mid \mathsf{RecvFromAny} \mid \mathsf{SendToQ} \mid \\ & & \mathsf{Ifq}\ instrs_1\ instrs_2 \mid \mathsf{GetIdleW} \mid \mathsf{UpdateWs} \end{array}$$

L'instruction SendTo envoie le contenu de la variable v au processus dont le *pid* apparaît en dernier dans $\mathcal{O}(p)$. L'instruction RecvFrom reçoit une donnée en provenance du processus dont le *pid* apparaît en premier dans $\mathcal{I}(p)$ et écrit cette donnée dans la variable iv. L'instruction Comp effectue un calcul en appelant un fonction séquentielle. L'argument de cette fonction est lu dans iv et son résultat écrit dans ov. L'instruction RecvFromAny attend (de manière non-déterministe) une donnée de l'ensemble des processus dont les *pid*s apparaissent dans $\mathcal{I}(p)$. La donnée reçue est placée dans la variable iv et le *pid* du processus émetteur est placé dans la variable q. L'instruction SendToQ envoie le contenu de la variable ov au processus dont le *pid* est donné dans la variable q. L'instruction Ifq compare la valeur contenue dans la variable q au *pid* listé en premier dans $\mathcal{I}(p)$. En cas d'égalité, la séquence d'instructions $instrs_1$ est exécutée ; sinon $instrs_2$ est exécutée. L'instruction UpdateWs lit la variable q et met à jour la variable iws en conséquence. La variable iws maintient la liste des processus esclaves libres pour un processus maître dans le cas d'un squelette de type FARM. L'instruction GetIdleW extrait un identificateur de processus de la liste iws et le place dans la variable q. Conjointement, ces deux dernières instructions encapsulent la stratégie utilisée au sein du squelette FARM pour assurer l'équilibre de charge.

On définit ensuite une algèbre simple permettant de construire, de manière compositionnelle des réseaux de processus. On utilisera pour cela les notations suivantes. Si $\mathcal{E}$ est un ensemble, on note $\mathcal{E}[e \leftarrow e']$ l'ensemble obtenu en remplaçant e par e' (en posant $\mathcal{E}[e \leftarrow e'] = \mathcal{E}$ si $e \notin \mathcal{E}$). Cette notation est supposée associative à gauche : $\mathcal{E}[e \leftarrow e'][f \leftarrow f']$ signifie $(\mathcal{E}[e \leftarrow e'])[f \leftarrow f']$. Si $e_1, \ldots, e_m$ est un sous-ensemble indexé de $\mathcal{E}$ et $\phi : \mathcal{E} \rightarrow \mathcal{E}$ une fonction, on notera $\mathcal{E}[e_i \leftarrow \phi(e_i)]_{i=1..m}$ l'ensemble $(\ldots((\mathcal{E}[e_1 \leftarrow \phi(e_1)])[e_2 \leftarrow \phi(e_2)])\ldots)[e_m \leftarrow \phi(e_m)]$. Sauf mention explicite, on notera $I(\pi_k) = \{i_k^1, \ldots, i_k^m\}$ et $O(\pi_k) = \{o_k^1, \ldots, o_k^n\}$. Par souci de concision, les listes $\mathcal{I}(o_k^j)$ et $\mathcal{O}(i_k^j)$ seront notées s_k^j et d_k^j respectivement. Pour les listes, on suppose définie l'opération de concaténation $++$ usuelle : si $l_1 = [e_1^1, \ldots, e_1^m]$ et $l_2 = [e_2^1, \ldots, e_2^n]$ alors $l_1{+}{+}l_2 = [e_1^1, \ldots, e_1^m, e_2^1, \ldots; e_2^n]$. La liste vide est notée $[]$. La longueur d'une liste l ou le cardinal d'un ensemble l sont tous deux notés $|l|$.

L'opérateur $\lceil . \rceil$ crée un réseau de processus contenant un seul processus à partir de son descripteur. La fonction NEW() fournit un nouveau *pid* à chaque appel.

$$\frac{\delta \in \Delta \qquad l = \textsc{New}()}{\lceil \delta \rceil = \langle \{(l, \langle [], [], \delta \rangle)\}, \{l\}, \{l\} \rangle} \quad (\textsc{Singl})$$

L'opérateur $\bullet$ "sérialise" deux réseaux de processus, en connectant les sorties du second aux entrées du premier :

$$\frac{\pi_i = \langle P_i, I_i, O_i \rangle \ (i = 1, 2) \qquad |O_1| = |I_2| = m}{\begin{array}{c}\pi_1 \bullet \pi_2 = \langle (P_1 \cup P_2)[(o_1^j, \sigma) \leftarrow \phi_d((o_1^j, \sigma), i_2^j)]_{j=1\ldots m}[(i_2^j, \sigma) \leftarrow \phi_s((i_2^j, \sigma), o_1^j)]_{j=1\ldots m}, \\ I_1, O_2 \rangle\end{array}} \quad (\textsc{Serial})$$

La règle précédente utilise deux fonctions auxiliaires ϕ_s et ϕ_d définies comme suit :

$\phi_s((p, \langle s, d, \langle \delta, \mathsf{Regular} \rangle \rangle), p') = (p, \langle [p']{+}{+}s, d, \langle [\mathsf{RecvFrom}]{+}{+}\delta, \mathsf{Regular} \rangle \rangle)$
$\phi_d((p, \langle s, d, \langle \delta, \mathsf{Regular} \rangle \rangle), p') = (p, \langle s, d{+}{+}[p'], \langle \delta{+}{+}[\mathsf{SendTo}], \mathsf{Regular} \rangle \rangle)$
$\phi_s((p, \langle s, d, \langle \delta, \mathsf{FarmM} \rangle \rangle), p') = (p, \langle [p']{+}{+}s, d, \langle \delta, \mathsf{FarmM} \rangle \rangle)$
$\phi_d((p, \langle s, d, \langle \delta, \mathsf{FarmM} \rangle \rangle), p') = (p, \langle s, d{+}{+}[p'], \langle \delta, \mathsf{FarmM} \rangle \rangle)$

La fonction ϕ_s (resp. ϕ_d) ajoute un processus p' aux prédécesseurs (resp. successeurs) d'un processus p et met à jour sa liste de macro-instructions. Concrètement, cela consiste à ajouter au début (resp. à la fin) de cette liste une instruction RecvFrom (resp. SendTo), sauf pour les processus maîtres au sein d'un squelette de type FARM, pour lesquels la liste d'instructions n'est pas modifiée (ces processus sont identifiés à l'aide du drapeau *kind*, positionné à la valeur FarmM).

L'opérateur $\|$ met deux réseaux de processus en parallèle en fusionnant leurs entrées et sorties respectivement.

$$\frac{\pi_i = \langle P_i, I_i, O_i \rangle \ (i = 1, 2)}{\pi_1 \parallel \pi_2 = \langle P_1 \cup P_2, I_1 \cup I_2, O_1 \cup O_2 \rangle} \quad (\textsc{Par})$$

L'opérateur $\bowtie$ relie deux réseaux de processus en connectant chaque sortie et sortie du second à la sortie du premier. Cet opérateur est utilisé pour décrire la structure cyclique qui correspond à une ferme de processus :

$$\frac{\pi_i = \langle P_i, I_i, O_i \rangle \ (i = 1, 2) \qquad |O_1| = 1 \qquad |I_2| = m \qquad |O_2| = n}{\begin{array}{c}\pi_1 \bowtie \pi_2 = \langle (P_1 \cup P_2)[(o_1, \sigma) \leftarrow \Phi((o_1, \sigma), I(\pi_2), O(\pi_2))][(i_2^j, \sigma) \leftarrow \phi_s((i_2^j, \sigma), o_1)]_{j=1\ldots m} \\ [(o_2^j, \sigma) \leftarrow \phi_d((i_2^j, \sigma), o_1)]_{j=1\ldots n}, \\ I_1, O_1 \rangle\end{array}} \quad (\textsc{Join})$$

où $\Phi(p, ps_s, ps_d) = \Phi_s(\Phi_d(p, ps_d), ps_s)$ et Φ_s (resp. Φ_d) est la généralisation de la fonction ϕ_s (resp. ϕ_d) à une liste de processus :

$\Phi_s(p, [p_1, \ldots, p_n]) = \phi_s(\ldots, \phi_s(\phi_s(p, p_1), p_2), \ldots, p_n)$
$\Phi_d(p, [p_1, \ldots, p_n]) = \phi_d(\ldots, \phi_d(\phi_d(p, p_1), p_2), \ldots, p_n)$

On peut dès lors expliciter la fonction $\mathcal{C}$ transformant un programme SKL en un réseau de processus :

$$
\begin{aligned}
\mathcal{C}[[\mathsf{Seq}\ f]] &= \lceil f \rceil \\
\mathcal{C}[[\mathsf{Pipe}\ \Sigma_1 \ldots \Sigma_n]] &= \mathcal{C}[[\Sigma_1]] \bullet \ldots \bullet \mathcal{C}[[\Sigma_n]] \\
\mathcal{C}[[\mathsf{Farm}\ n\ \Sigma]] &= \lceil \mathsf{FarmM} \rceil \bowtie (\mathcal{C}[[\Sigma]]_1 \parallel \ldots \parallel \mathcal{C}[[\Sigma]]_n) \\
\mathcal{C}[[\mathsf{Pardo}\ \Sigma_1\ \ldots \Sigma_n]] &= \mathcal{C}[[\Sigma_1]] \parallel \ldots \parallel \mathcal{C}[[\Sigma_n]]
\end{aligned}
$$

où FarmM est le descripteur encapsulant le comportement d'un processus maître au sein d'un squelette FARM :

$$\Delta(\mathsf{FarmM}) = \langle [\mathsf{RecvFromAny}; \mathsf{Ifq}\ [\mathsf{GetIdleW}; \mathsf{SendToQ}]\ [\mathsf{UpdateWs}; \mathsf{SendTo}]], \mathsf{FarmM} \rangle$$

4.2. Implémentation en MetaOcaml

L'intégration ("*embedding*") du langage SKL au langage hôte (METAOCAML donc ici) se fait via une fonction dite d'interprétation (`run`) qui se charge de convertir une spécification SKL en un code exécutable, appelé *code résiduel*, sur chaque processeur.

Concrètement, le programme exemple donné au paragraphe précédent s'écrira alors de la manière suivante :

```
let f () = (* code de la fonction séquentielle générant les données à traiter *)
let g x = (* code de la fonction séquentielle de traitement des données *)
let h x = (* code de la fonction séquentielle de traitement des résultats *)
let pgm =  Pipe [seq f; Farm(3, seq g); seq h]
let _ = run pgm
```

Le programme SKL est ici décrit via un type algébrique récursif :

```
type skl_tree =
    Seq of seq_comp
  | Pipe of skl_tree list
  | Pardo of skl_tree list
  | Farm of int * skl_tree
```

La définition exacte du type `seq_comp` sera donnée plus loin. Il suffit de dire pour l'instant que l'on dispose d'une fonction d' "encapsulation" `seq` permettant de transformer toute fonction séquentielle `f : 'a->'b` en une valeur de type `seq_comp`.

La génération du code résiduel par la fonction `run` se fait en deux temps :

1. la représentation arborescente du programme parallèle est d'abord transformée en une représentation de ce même programme sous la forme d'un réseau de processus séquentiels communiquants (RPSC)
2. puis, le code résiduel est généré en confrontant cette représentation intermédiaire – au sein de laquelle chaque processus est repéré par un *pid* unique – avec le rang effectif du processeur sur lequel la fonction d'interprétation `run` s'exécute ; en d'autres termes ne sera généré sur le processeur i que le code correspondant au processus ayant le *pid* i au sein du RPSC.

```
type process_network = {
    procs: (pid * process_desc) list;
    inputs: pid list;
    outputs: pid list;
  }

and process_desc = {
    kind: process_kind;
    instrs: instr list;
    recvfrom: pid list;
    sendto: pid list;
  }

and process_kind = Regular | FarmM
```

```
and instr =
  | Comp of seq_comp
  | SendTo
  | RecvFrom
  | RecvFromAny
  | SendToQ
  | Ifq of instr list * instr list
  | GetIdleW
  | UpdateWs

and seq_comp = process_state -> unit

and process_state = {
    mutable iv: seq_val;
    mutable ov: seq_val;
    mutable q: pid;
    mutable iws: pid list
  }
```

FIG. 3 – Listing 1

4.2.1. Génération du réseau de processus

Le listing 1 donne la traduction des principaux types de données utilisés pour cette étape.

Les types de `process_network`, `process_desc`, `process_kind` et `instr` découlent immédiatement des définitions données à la section 4.1 et n'appellent pas de remarque particulière. Les points intéressants concernent les types `process_state` et `seq_comp`.

Le type `process_state` regroupe les quatre variables manipulées par les macro-instructions et le type `seq_comp` traduit désormais que l'effet des fonctions de calcul séquentielles est de mettre à jour l'état d'un processus. En pratique, une telle fonction sera appelée avec comme argument le contenu de la variable `iv` et son résultat sera écrit dans la variable `ov`[5].

Typage. Une difficulté se pose toutefois lorsque l'on cherche à préciser le type `seq_val` ; en effet, le type des composantes `iv` et `ov` dépend *in fine* du type de la fonction séquentielle de calcul exécutée par le processus. En supposant que le type de cette fonction soit `'a->'b`, on pourrait imaginer paramétrer le type `process_state` :

```
type ('a,'b) process_state = {
    mutable iv: 'a;
    mutable ov: 'b;
    ... }
```

Mais dans ce cas, le type `process_network` devient `('a,'b) process_network` et il est impossible de définir des réseaux dont les processus opèrent sur des données de types différents[6].

Dans le contexte des DSLs métaprogrammés, ce problème est bien connu et apparaît chaque fois que les langages objet et hôte sont statiquement typés (comme ici). La solution consiste en général

[5] Une fonction séquentielle de calcul doit donc avoir un type de la forme `'a -> 'b`. On peut toujours de se ramener, par décurryfication si besoin, à cette forme de signature.

[6] Ce problème est analogue à celui auquel on se heurte classiquement lorsque l'on cherche à définir une fonctionnelle `compose` généralisée : si on écrit `let rec gen_compose = function [f] -> f | f : :fs -> compose f (gen_compose fs)`, avec `let compose f g = function x -> g (f x)` alors le type inféré est `('a -> 'a) list -> ('a -> 'a)` ce qui oblige toutes les fonctions a avoir la même signature.

à introduire un type « universel » encapsulant de manière uniforme tous les types susceptibles d'être manipulés par les fonctions séquentielles de calcul du programme. Dans notre cas, on définit alors le type `seq_val` comme :

```
type seq_val = Int of int | Float of float | IntArray of int array | ...
```

et un *wrapper* assure l'étiquetage (resp. « dés-étiquetage ») des données en provenance des (resp. passées aux) fonctions séquentielles de calcul. Par exemple, si `f` est une fonction de type `int -> float`, on définit :

```
let f' = function (Int x) ->  Float (f x)
```

et c'est cette fonction qui est passée au constructeur `Seq` afin de construire les programmes Skl :

```
let seq f = Seq (fun s -> s.ov <- f s.iv)
```

Remarque. Cette solution oblige évidemment à créer un *wrapper* spécifique pour chaque fonction séquentielle et à recompiler l'interpréteur à chaque fois que des types nouveaux sont requis. Elle induit par ailleurs un surcoût à l'exécution. On peut éviter ceci en contournant les règles du typage statique d'Ocaml au moyen des primitives `Obj.repr` et `Obj.obj`. On on a alors simplement :

```
type seq_val = Obj.repr

let (seq : ('a -> 'b) -> seq_comp) =
  function f -> Seq (fun s -> s.ov <- Obj.repr (f (Obj.obj s.iv)))
```

Mais ceci se fait évidemment au détriment de la sûreté puisque le contrôle de type est alors *de facto* inhibé. Par exemple, un programme comme `Pipe [sql f; sql g]`, où `f:int->float` et `g:int->int`, est accepté et déclenche évidemment une erreur à l'exécution.

En pratique, nous disposons de deux versions de l'interpréteur : la première, utilisant la définition « universelle » de `seq_val` permet de vérifier la cohérence des types dans un programme Skl. La seconde, utilisant `Obj.repr`, permet de supprimer l'*overhead* lié au *tagging* dynamique des valeurs.

Les opérateurs $\lceil . \rceil$, $\bullet$, $\|$ et $\bowtie$ sont implantés par traduction directe des règles de productions de la sémantique sur la base des types définis ci-dessus. On ne reproduira donc ici que le code associé aux deux premiers :

```
let singl d =
  let p = new_pid () in
  { procs = [p, d]; inputs = [p]; outputs = [p] }

let serial pn1 pn2 =
  if List.length pn1.outputs = List.length pn2.inputs then
    { procs = List.map
        (function (pid, pd) ->
          if List.mem pid pn1.outputs then
            let dst = List.assoc pid (List.combine pn1.outputs pn2.inputs) in
            (pid, add_dst pd dst)
          else if List.mem pid pn2.inputs then
            let src = List.assoc pid (List.combine pn2.inputs pn1.outputs) in
            (pid, add_src pd src)
```

```
          else (pid, pd))
        (pn1.procs @ pn2.procs);
      inputs = pn1.inputs;
      outputs = pn2.outputs }
  else failwith "cannot serialize process networks: size mismatch"
```

Dans ce qui précède, les fonctions `add_src` et `add_dst` implémentent les fonctions ϕ_s et ϕ_d. Par exemple :

```
let add_dst pd1 pid2 =
  match pd1.kind with
    Regular -> {pd1 with instrs = pd1.instrs @ [SendTo]; sendto = pd1.sendto @ [pid2]}
  | FarmM -> {pd1 with sendto = pd1.sendto @ [pid2]}
```

Enfin, la fonction `expand_tree` implémente la fonction de conversion $\mathcal{C}$ introduite à la section 4.1 par simple filtrage sur le type récursif représentant les programmes Skl :

```
let (expand_tree : skl_tree -> process_network) = function t ->
  let rec expand = function
      Seq f -> singl {kind=Regular; instrs=[Comp f]; recvfrom=[]; sendto=[]; workers=[]}
    | Pipe [] -> failwith "empty pipe"
    | Pipe [t] -> expand t
    | Pipe (t::ts) ->
        serial (expand t) (expand (Pipe ts))
    | Pardo [] -> failwith "empty pardo"
    | Pardo [t] -> expand t
    | Pardo (t::ts) ->
        par (expand t) (expand (Pardo ts))
    | Farm (n, t) ->
        if n > 0 then
          let workers =
            let rec create n =
              if n=1 then expand t
              else
                par (expand t) (create (n-1)) in
            create n in
          join (farmer workers.inputs) workers
        else
        failwith "cannot expand farm with n<=0 workers"
  in
  reset_pid ();
  expand t

let farmer ws = singl {
  kind=FarmM;
  instrs=[RecvFromAny; Ifq ([GetIdleW; SendToQ],[UpdateWs; SendTo])];
  recvfrom=[]; sendto=[]; workers=ws }
```

4.2.2. Génération du code résiduel

La fonction `code` génère, à partir de la représentation du réseau de processus calculée à l'étape précédente, le code résiduel sur un processeur de rang donné :

```
let code pn rank =
  let mydesc = List.assoc rank pn.procs in
 .< while true do .~(code_instrs mydesc mydesc.instrs) done >.
```

Le code est "assemblé" par simple parcours de la liste de macro-instructions associée au descripteur du processus. C'est à ce niveau que la variable d'état du processus est créée[7]. Le contenu initial des composantes *iv*, *ov* et *q* est indifférent. Les processus de type *FarmM* voient leur composante *iws* initialisée avec la liste des processus esclaves auxquels ils sont reliés :

```
let code_instrs pd is =
  let s = { iv=Obj.repr 0; ov=Obj.repr 0; q=0; iws=pd.workers } in
  List.fold_left
    (fun c i -> .< begin .~c; .~(code_instr pd s i) end >.)
    .< () >.
    is
```

Enfin la fonction `code_instr` génère le code correspondant à chaque macro-instruction :

```
let rec code_instr pd s = function
    Comp f -> .< f s >.
  | SendTo ->
      let dst = List.hd (List.rev pd.sendto) in
     .< Mpi.send s.ov dst 0 Mpi.comm_world  >.
  | RecvFrom ->
      let src = List.hd (pd.recvfrom) in
     .< s.iv <- Obj.repr (Mpi.receive src Mpi.any_tag Mpi.comm_world) >.
  | RecvFromAny ->
     .< let r,q,_ = Mpi.receive_status Mpi.any_source Mpi.any_tag Mpi.comm_world in
        s.ov <- Obj.repr r; s.q <- q >.
  | SendToQ ->
     .< Mpi.send s.ov s.q 0 Mpi.comm_world >.
  | Ifq (is1,is2)->
     .< if s.q = List.hd pd.recvfrom
        then .~(code_instrs pd is1) else .~(code_instrs pd is2) >.
  | GetIdleW ->
      .< begin s.q <- List.hd s.iws; s.iws <- List.tl s.iws end >.
  | UpdateWs ->
      .< s.iws <- s.iws @ [s.q] >.
```

Une instruction Comp f produit un appel à la fonction séquentielle f. Les instructions SendTo, SendToQ, RecvFrom et RecvFromAny sont traduites directement en appel aux primitives MPI (on utilise ici la bibliothèque OcamlMpi [18]). Pour l'instruction Ifq l'opérateur d'échappement de MetaOcaml permet d'insérer directement dans le code produit les fragments correspondant aux deux branches de la conditionnelle. Concernant les instructions GetIdleW et UpdateWs, on s'est contenté ici d'implanter une simple gestion du *pool* d'esclaves en tourniquet en manipulant la liste *iws* (*idle workers*) en FIFO (retrait en tête par l'instruction GetIdleW, insertion en queue par l'instruction UpdateWs). D'autres stratégies de gestion sont évidemment envisageables.

La fonction d'interprétation principale s'écrit alors aisément :

[7] On tire parti ici de la possibilité offerte par MetaOcaml d'utiliser une variable définie au niveau i aux niveaux $i+n$ ("cross-stage persistence").

```
let size = Mpi.comm_size Mpi.comm_world
let myrank = Mpi.comm_rank Mpi.comm_world

let run pgm =
   let pn = expand_tree pgm  in
   let mycode = code pn myrank in
   .! (mycode)
```

A titre d'exemple, voici le code résiduel produit sur chaque processeur pour le programme `Pipe[seq f; seq g; seq h]` :

PID	Code
2	`.< while true do f s; Mpi.send s.ov 1 0 Mpi.comm_world done >.`
1	`.< while true do s.iv <- Obj.repr (Mpi.receive 2 0 Mpi.comm_world); g s; Mpi.send s.ov 0 0 Mpi.comm_world done >.`
0	`.< while true do s.iv <- Obj.repr (Mpi.receive 1 0 Mpi.comm_world); h s done >.`

Ce code est identique à celui qu'aurait écrit un programmeur MPI expérimenté.

5. Résultats

Nous avons évalué l'impact de cette technique d'implémentation en mesurant le surcoût (*overhead*) ρ en temps d'exécution qu'elle introduit par rapport à un code parallèle écrit directement à l'aide de primitives MPI (Ocaml + bibliothèque OcamlMPI v 1.0.1)[8].

Ce surcoût a été mesuré pour deux types de programmes : des programmes ne mettant en jeu qu'un seul squelette et des programmes pour lesquels plusieurs squelettes sont imbriqués à une profondeur arbitraire. Pour le premier type de test, on observe l'effet de deux paramètres : la durée d'exécution τ de la fonction de calcul séquentielle et la "taille" N du squelette (cette taille correspond au nombre d'étages pour un squelette PIPE, au nombre de tâches parallèles pour un squelette PARDO et au nombre d'esclaves pour un squelette FARM). Pour le second type de test, nous nous sommes limités au cas du squelette FARM. On évalue alors les performances d'un programme formé de P squelettes FARM imbriqués, chacun mettant en jeu ω esclaves.

Les résultats ont été obtenus sur un *cluster* à 30 noeuds de type PowerPC G5 pour N variant entre 2 et 30 et $\tau = 1ms, 10ms, 100ms, 1s$.

Pour le squelette PIPE, ρ n'excède jamais 2 %. Pour FARM et PARDO, ρ reste inférieur à 3 % et devient négligeable pour $N > 8$ ou $\tau > 10ms$. En cas d'imbrication, le pire cas est obtenu pour $P = 4$ et $\omega = 2$; ρ varie alors entre 7 % et 3 % quand τ passe de $1ms$ à $1s$.

Ces résultats sont bien meilleurs que ceux obtenus avec des systèmes de programmation par squelettes n'exploitant pas la métaprogrammation. Dans l'implémentation décrite dans [13], par exemple, où les squelettes sont implantées comme de simples fonctions d'ordre supérieur ordonnançant dynamiquement (c-à-d. *à l'exécution*) des primitives MPI, le surcoût était de l'ordre de 10 à 20 %. Pour la dernière version du système SKIPPER [14], dans laquelle les squelettes étaient traduits en graphes flots de données exécutés par un interpréteur dédié, ce surcoût atteignait parfois 100 %. D'autres implantations, comme celle de Kuchen [7], dans laquelle les squelettes sont traduits par des appels de méthodes virtuelles en C++, font aussi état de surcoût à l'exécution – par rapport à du code C+MPI dans ce cas - de l'ordre de 20 à 120 %.

[8]En utilisant la version « optimisée » de l'interpréteur, *i.e.* celle exploitant les primitives `Obj.repr` et `Obj.obj`.

6. Travaux connexes

L'idée d'exploiter la métaprogrammation pour implanter de manière efficace un DSL n'est pas neuve. Le principe sous-jacent consiste en transformer par *évaluation partielle* un couple (interpréteur de ce langage, programme dans ce langage) en un programme compilé . Les techniques de programmation dite *générative* [6] qui en découlent visent en général à résoudre la tension entre abstraction et performances, comme dans notre cas. Les bibliothèques FFTW [4], EVE [3] et SPIRAL [15] peuvent être vues comme des illustrations de cette approche.

Les langages fonctionnels offrent un substrat favorable pour la métaprogrammation grâce en particulier aux fonctions d'ordre supérieur et à la possibilité de coder très facilement la syntaxe abstraite du DSL sous la forme d'un type algébrique du méta-langage. Dans [11], Sheard fait une revue des techniques de métaprogrammation dans le contexte de la programmation fonctionnelle. Il y introduit notamment le langage MetaML qui ajoute au langage ML des annotations permettant de transformer des expressions ML en fragment de code, d'insérer des fragments de code dans d'autres fragments de code et d'exécuter des fragments de code. Le langage MetaOcaml [20] développé par Taha *et al.* est une extension similaire du langage Ocaml. Le langage *Template Haskell* [12] ajoute, de son coté, un large panel de facilités pour la métaprogrammation au langage *Haskell*, avec notamment des possibilités pour le méta-langage d'accéder à sa propre syntaxe abstraite (instrospection). En revanche, et contrairement à MetaML et MetaOcaml, la génération du code résiduel a lieu à la compilation et ne peut donc pas prendre en compte des données dynamiques.

Les problématiques générales liées à la définition d'un DSL pour le parallélisme sont étudiées par Lengauer dans [8]. Mais, dans ce domaine, seul Herrmann, dans [5] semble avoir exploré les possibilités offertes par la génération dynamique de code avec MetaOcaml pour implanter un tel DSL. Le système qu'il décrit est similaire à celui présenté ici mais en diffère toutefois en plusieurs points. Premièrement, le modèle de parallélisme sur lequel il repose est restreint aux schémas fixes et déterministes (structures série-parallèle) ; il est donc impossible d'y exprimer des squelettes pour lesquels l'ordonnancement des communications n'est pas connu à la compilation (comme farm). Deuxièmement, dans l'approche décrite, les squelettes ne font pas à proprement partie du DSL mais sont définis à partir du vocabulaire de ce dernier. Enfin, la sémantique du langage n'est pas définie formellement mais directement encodée dans l'interpréteur en MetaOcaml (il n'y pas de représentation intermédiaire explicite sous la forme de réseau de processus communiquants en particulier).

7. Conclusion

Nous avons montré dans cet article comment les facilités de métaprogrammation offertes par le langage MetaOcaml permettent d'implanter efficacement un DSL dédié à la programmation parallèle en conciliant haut niveau d'abstraction et performances. Le fait que le code résiduel soit généré à l'exécution est ici un avantage dans la mesure où cette génération peut prendre en compte la configuration effective de la machine (nombre de processeurs, ...). Bien sur le temps requis pour générer ce code s'ajoute au temps d'exécution de l'application elle-même mais, pour des problèmes de taille suffisante, il reste négligeable.

Le prototype décrit ici ne génère que du *bytecode*, le compilateur natif de MetaOcaml n'étant pas disponible sur toutes les plateformes. Dans la pratique ce point n'est pas critique dans la mesure où il est parfaitement possible d'appeler des fonctions de calcul séquentielles écrites en C, C++ ou Fortran en s'appuyant sur la *Foreign Function Interface* d'Ocaml. Une autre approche consisterait à utiliser les facilités d'*offshoring* offertes par MetaOcaml, c-à-d. la possibilité de générer automatiquement du code C à partir de code Ocaml.

Le principal problème à régler dans notre implémentation concerne toutefois le typage. La solution retenue, avec deux versions de l'interpréteur, n'est pas théoriquement satisfaisante. Il reste donc à voir

dans quelle mesure certaines techniques d'élimination automatique des étiquettes de typage, décrites par exemple dans [16], pourraient s'appliquer.

A plus long terme nous envisageons d'appliquer la techniques présentée ici à la réimplémentation d'autres systèmes de programmation parallèle fondés sur les squelettes, comme Skipper[14] ou OcamlP3L[19].

Références

[1] B. Bacci, M. Danelutto, S. Orlando, S. Pelagatti, and M. Vanneschi. P3L : A Structured High Level Programming Language And Its Structured Support. *Concurrency : Practice and Experience*, pages 225–255, 1995.

[2] M. Cole. Bringing Skeletons out of the Closet : A Pragmatic Manifesto for Skeletal Parallel Programming. *Parallel Computing*, 3 :389–406, 2004.

[3] J. Falcou and J. Sérot. EVE : an object-oriented SIMD library. *Scalable Computing : Practice and Experience*, 6(4) :31–42, 2005.

[4] Kang Su Gatlin and Larry Carter. Faster FFTs via architecture-cognizance. In *IEEE PACT*, pages 249–260, 2000.

[5] Christoph A. Herrmann. Generating message-passing programs from abstract specifications by partial evaluation. *Parallel Processing Letters*, 15(3) :305–320, 2005.

[6] Neil D. Jones and Arne J. Glenstrup. Program generation, termination, and binding-time analysis. In *ICFP '02 : Proceedings of the seventh ACM SIGPLAN international conference on Functional programming*, pages 283–283, New York, NY, USA, 2002. ACM Press.

[7] H. Kuchen. A skeleton library. *Proceedings of the 8th International Euro-Par Conference on Parallel Processing*, pages 620–629, 2002.

[8] Christian Lengauer, Don S. Batory, Charles Consel, and Martin Odersky, editors. *Domain-Specific Program Generation, International Seminar, Dagstuhl Castle, Germany, March 23-28, 2003, Revised Papers*, volume 3016 of *Lecture Notes in Computer Science*. Springer, 2004.

[9] G. Michaelson N. Scaife and S. Horiguchi. Parallel Standard ML with Skeletons. *Scaleable Computing Practise and Experience*, 6(4), 2006.

[10] J. Sérot and D. Ginhac. Skeletons for parallel image processing : an overview of the SKiPPER project. *Parallel Computing*, 28(12) :1785–1808, Dec 2002.

[11] Tim Sheard. Accomplishments and research challenges in meta-programming. In *SAIG 2001 : Proceedings of the Second International Workshop on Semantics, Applications, and Implementation of Program Generation*, pages 2–44, London, UK, 2001. Springer-Verlag.

[12] Tim Sheard and Simon Peyton Jones. Template metaprogramming for Haskell. In Manuel M. T. Chakravarty, editor, *ACM SIGPLAN Haskell Workshop 02*, pages 1–16. ACM Press, October 2002.

[13] J. Sérot Embodying parallel functional skeletons : an experimental implementation on top of MPI. *3rd Intl Euro-Par Conference on Parallel Processing*, Aout 1997, Passau. Volume 1300 of LNCS, pp 629–633, Springer.

[14] J. Sérot. Tagged-token data-flow for skeletons. *Parallel Processing Letters*, 11(4) :377-392, 2002.

[15] José M. F. Moura, Jeremy Johnson, Robert W. Johnson, David Padua, Viktor K. Prasanna, Markus Püschel, Bryan Singer, Manuela Veloso, and Jianxin Xiong. Generating platform-adapted dsp libraries using SPIRAL. In *High Performance Embedded Computing (HPEC)*, 2001.

[16] Emir Pašalic, Walid Taha, Tim Sheard. Tagless staged interpreters for typed languages. *SIGPLAN Not.*, 37(9) : :218–229, 2002.

[17] `http ://caml.inria.fr`

[18] `http ://caml.inria.fr/cgi-bin/hump.en.cgi ?contrib=401`

[19] `http ://ocamlp3l.inria.fr`

[20] `http ://www.metaocaml.com`