Le filtrage passé au crible

Paul Patault*

21 juin 2022

Résumé

GOSPEL est un langage logique qui permet de spécifier formellement du code OCaml. Ce langage inclut naturellement une construction `match-with` de filtrage par motifs. Dans ce travail, nous présentons un algorithme pour vérifier l'exhaustivité de la construction de filtrage et une preuve de correction de cet algorithme. Nous avons implémenté cet algorithme dans le langage GOSPEL et pour augmenter encore notre confiance dans cette implémentation, nous avons automatisé une génération de problèmes de filtrage de taille arbitraire. Ceci nous a notamment permis de vérifier notre programme en utilisant le compilateur OCaml comme une implémentation de référence.

1 Introduction

Le langage GOSPEL [3] est un langage de spécification formelle pour le langage de programmation OCaml conjointement développé entre le Laboratoire des Méthodes Formelles et l'entreprise Tarides. Permettant d'annoter les interfaces OCaml, avec pour chaque fonction un contrat formel, le langage GOSPEL est notamment utilisé pour faire de la vérification à l'exécution [6] ou de la vérification déductive [16].

Le sujet de ce stage, encadré par Jean-Christophe Filliâtre [1] et Clément Pascutto [2], est d'étendre le langage GOSPEL en y ajoutant notamment un algorithme de vérification d'exhaustivité pour la construction du filtrage ainsi que l'ajout des clauses `when` [3]. Le filtrage par motifs (de l'anglais *pattern matching*), est une construction fondamentale des langages de programmation fonctionnels. Permettant de raisonner par cas, aussi bien sur la valeur d'un entier que la structure d'un type algébrique défini par le programmeur, il s'avère intéressant d'automatiser la tâche de vérification d'exhaustivité de notre ensemble de motifs. Retrouvez ci-dessous deux exemples de cas d'utilisation du filtrage dans le langage OCaml :

```
let rec fibo = function 0 -> 0 | 1 -> 1 | n -> fibo (n-1) + fibo (n-2)
let rec mem42 = function [] -> false | 42::_ -> true | _::t -> mem42 t
```

Nos contributions sont l'implémentation dans GOSPEL d'un algorithme de vérification d'exhaustivité de la construction de filtrage, une preuve de correction de cet algorithme, ainsi que la conception d'un générateur de problèmes de filtrage de taille arbitraire (ce qui a permis de tester sérieusement l'implémentation). De plus, nous avons ajouté les clauses `when` qui n'étaient pas présentes initialement dans GOSPEL et nous avons trouvé une transformation permettant d'étendre l'algorithme d'exhaustivité pour certains filtrages contenant des `when`.

*MPRI, paul.patault@universite-paris-saclay.fr

1. LMF, jean-christophe.filliatre@cnrs.fr

2. Tarides, clement@tarides.com

3. Cette recherche a été [partiellement] soutenue par le Labex DigiCosme (projet ANR11LABEX0045DIGICOSME) opéré par l'ANR dans le cadre du programme « Investissement d'Avenir » Idex ParisSaclay (ANR11IDEX000302).

En section 2.1, nous commençons par formaliser le problème de l'exhaustivité, puis en section 2.3 nous présentons l'algorithme résolvant ce problème. Ensuite, nous faisons la preuve de correction totale en sections 2.5 et 2.6, suivi d'une analyse de borne inférieure pour la complexité en section 2.7. Puis, en section 2.9 nous discutons de l'implémentation concrète dans GOSPEL ainsi que des tests réalisés. Enfin, nous présentons l'intégration dans le travail réalisé des clauses `when`, des problèmes posés par celle-ci et des solutions proposées.

2 Problème du motif utile

2.1 Formalisation

Les motifs que nous traitons se déclinent sous différentes formes.

Définition 1 (Motifs)**.** Soit $\mathbb{M}$ l'ensemble des motifs, décrit par la syntaxe abstraite suivante :

m	::=			
	$\mid$ `_`	*Attrape-tout*	:	`_`
	$\mid$ x	*Variable*	:	`x`
	$\mid$ $C(m, \ldots, m)$	*Construction*	:	`C (1, _)` / `42` / `"tango"`
	$\mid$ $m \mid m$	*Motif-ou*	:	`A | B`
	$\mid$ m as x	*Motif-as*	:	`(x::s) as l`
	$\mid$ $c..c$	*Intervalle*	:	`'a'..'z'` / `'b'..'b'`

Remarque 1. Certains de ces motifs se retrouvent souvent regroupés, en particulier dans l'algorithme que nous allons présenter. Exemplairement, les motifs attrape-tout / variables seront toujours interchangeables. Remarquons aussi que les constantes (*i.e.* entiers, chaînes de caractères, ...) sont analogues à des constructeurs d'arité 0. D'un autre côté, les n-uplets sont des constructions avec pour chaque dimension un constructeur implicite particulier (*e.g.* `Tuple2`). Enfin, les motifs-as sont toujours effacés tels que `m as x` est traité comme `m`.

Définition 2 (Types)**.** Les motifs peuvent être typés. Un type τ est défini par l'expression de type :

T	::=	$int \mid bool \mid char \mid list \mid option \mid \ldots$	*symbole de type*
τ	::=		
	$\mid$ α		*variable de type*
	$\mid$ $(\tau, \ldots, \tau)\ T$		*type paramétré*

Définition 3 (Motif linéaire)**.** On dit qu'un motif m est linéaire si toute variable apparaît au plus une fois dans m.

Définition 4 (Motif bien typé)**.** On dit qu'un motif m est de type τ, noté $m : \tau$, si m est linéaire et qu'il respecte les règles de typage suivante :

$$\frac{}{\_ : \tau}\ (\textsc{Def}) \qquad \frac{}{x : \tau}\ (\textsc{Var}) \qquad \frac{}{c_1..c_2 : char}\ (\textsc{Inter}) \qquad \frac{m_1 : \tau \qquad m_2 : \tau}{m_1 \mid m_2 : \tau}\ (\textsc{Or})$$

$$\frac{m : \tau}{m \text{ as } x : \tau}\ (\textsc{As}) \qquad \frac{\texttt{type}\ \vec{\alpha}\ T = C\ of\ \tau_1 \times \cdots \times \tau_k \mid \ldots \qquad \forall i.\ i \in [1..k] \Rightarrow m_i : \tau_i[\vec{\alpha}/\vec{\tau}]}{C(m_1, \ldots, m_k) : \vec{\tau}\ T}\ (\textsc{Constr})$$

Définition 5 (Taille d'un motif)**.** Nous définissons la taille d'un motif par :

$$\begin{aligned} \mathit{taille}(\_) &= 0 \\ \mathit{taille}(\texttt{x}) &= 0 \\ \mathit{taille}(c_1..c_2) &= 1 \\ \mathit{taille}(m_1 \mid m_2) &= 1 + \mathit{taille}(m_1) + \mathit{taille}(m_2) \\ \mathit{taille}(C(m_1, \ldots, m_k)) &= 1 + \textstyle\sum_{i=1}^{k} \mathit{taille}(m_k) \\ \mathit{taille}(m \texttt{ as x}) &= 1 + \mathit{taille}(m) \end{aligned}$$

Définition 6 (Relation de filtrage)**.** Soit $(m, m') \in \mathbb{M} \times \mathbb{M}$, on dit que le motif m filtre le motif m', noté $m \preceq m'$, si m est plus général (ou moins précis) que m'. Cette relation peut se définir par les règles suivantes :

$$\frac{}{\_ \preceq m}\ (\textsc{Def}) \qquad \frac{}{x \preceq m}\ (\textsc{Var}) \qquad \frac{m \preceq m'}{m \text{ as } x \preceq m'}\ (\textsc{As})$$

$$\frac{m \preceq m'_1 \qquad m \preceq m'_2}{m \preceq m'_1 \mid m'_2}\ (\textsc{Or}_0) \qquad \frac{m_1 \preceq m'}{m_1 \mid m_2 \preceq m'}\ (\textsc{Or}_1) \qquad \frac{m_2 \preceq m'}{m_1 \mid m_2 \preceq m'}\ (\textsc{Or}_2)$$

$$\frac{c_1 \leq c'_1 \qquad c'_2 \leq c_2}{c_1..c_2 \preceq c'_1..c'_2}\ (\textsc{Inter}) \qquad \frac{\forall i.\ i \in [1..k] \Rightarrow m_i \preceq m'_i}{C(m_1, \ldots, m_k) \preceq C(m'_1, \ldots, m'_k)}\ (\textsc{Constr})$$

Définition 7 (Matrice de filtrage)**.** On dit que M est une matrice de filtrage si $M \in \mathcal{M}_{m \times n}(\mathbb{M})$. Cette matrice M correspond à un filtrage de m lignes sur un n-uplet en OCaml. Nous pouvons illustrer cette correspondance par l'exemple suivant :

```
let f = function
| x,      Nil            -> 1
| Some _, Cons (42, Nil) -> 2
| _                      -> 3
```

$$\begin{pmatrix} \texttt{x} & \texttt{Nil} \\ \texttt{Some \_} & \texttt{Cons(42, Nil)} \\ \texttt{\_} & \texttt{\_} \end{pmatrix}$$

Remarque 2. L'attrape-tout présent en ligne 3 du filtrage est « dupliquable » autant de fois que nécessaire pour obtenir la bonne dimension dans la matrice.

Remarque 3. La structure matricielle est un point clé de la conception de l'algorithme que nous présenterons en partie 2.3. C'est pourquoi, même si l'on filtre uniquement des 1-uplets, nous utilisons cette représentation par matrice.

Définition 8 (Matrice bien formée)**.** Une matrice $M \in \mathcal{M}_{m \times n}(\mathbb{M})$ est bien formée pour un type $\vec{\tau}$, noté $M : \vec{\tau}$, si et seulement si chaque ligne de la matrice est du type $\vec{\tau}$.

$$\frac{\forall i \in [1..m].\ \forall j \in [1..n].\ M_{i,j} : \tau_j}{M_{m \times n} : \vec{\tau}}\ (\textsc{Mat})$$

Définition 9 (Filtrage par matrice)**.** On dit que la matrice M filtre le vecteur $\vec{q}$, noté $M \preceq \vec{q}$, si et seulement si chaque composante d'une même ligne de M filtre la composante correspondante dans $\vec{q}$.

$$\frac{\exists i \in [1..m].\ \forall j \in [1..n].\ M_{i,j} \preceq q_j \qquad M : \tau \qquad \vec{q} : \tau}{M_{m \times n} \preceq \vec{q}}\ (\textsc{Mat})$$

Définition 10 (Taille d'une matrice)**.** Soit $M \in \mathcal{M}_{m \times n}(\mathbb{M})$ une matrice de filtrage, la taille de M est définie par la somme des tailles des motifs qui la compose.

$$\mathit{taille}(M) = \sum_{i=1}^{m} \sum_{j=1}^{n} \mathit{taille}(M_{i,j})$$

2.2 Problème de l'exhaustivité

Les notions d'exhaustivité et de motif utile sont exprimables dans le cadre des matrices de filtrage.

Définition 11 (Exhaustivité)**.** Soit $M \in \mathcal{M}_{m\times n}(\mathbb{M}) : \vec{\tau}$ une matrice de filtrage. Cette matrice est dite exhaustive si et seulement si pour tout vecteur de motifs $\vec{q} \in \mathbb{M}^n : \vec{\tau}$, il existe une ligne de M qui filtre q. C'est-à-dire :

$$\forall \vec{q} \in \mathbb{M}^n.\ \vec{q} : \vec{\tau} \Rightarrow M \preceq \vec{q}$$

Définition 12 (Motif utile)**.** Un vecteur de motifs $\vec{q} \in \mathbb{M}^n$ est dit utile relativement à une matrice $M \in \mathcal{M}_{m\times n}(\mathbb{M})$ si et seulement si $\vec{q}$ et M sont de même type et qu'aucune des m lignes de M ne filtre $\vec{q}$.

2.3 Résolution du problème

La question à laquelle on souhaite répondre est donc : une matrice bien formée est-elle exhaustive ? Pour résoudre ce problème, nous utiliserons l'algorithme `usefulness` (Luc Maranget 2007 [13]) dont la signature est `(mat` : τ`) -> (vec` : τ`) -> bool`. À partir d'une matrice de filtrage M et d'un vecteur de motifs q, `usefulness`$(M, \vec{q})$ renvoie vrai si et seulement si le vecteur $\vec{q}$ est utile à M au sens de la définition (12).

Ainsi, nous pouvons répondre à la question posée en appelant l'algorithme `usefulness` sur le vecteur $\vec{q} = (\_, \ldots, \_)$ et la matrice testée est exhaustive si et seulement si le résultat renvoyé est faux.

Notation 1 (Arité)**.** Nous utiliserons « $|C|$ » pour indiquer l'arité du constructeur C.

Notation 2 (Suite de la ligne)**.** Pour une matrice de filtrage M, nous désignerons par « `tail`$_i$ » la i-ème ligne, en excluant son premier élément : `tail`$_i = M_{i,2} \cdots M_{i,n}$.

Nous partons d'une matrice que nous allons réduire au fur et à mesure à l'aide d'un algorithme récursif, en raisonnant par cas sur les motifs de la première colonne. Définissons dans un premier temps deux fonctions auxiliaires `default` et `spec` dont nous aurons besoin par la suite. Ces deux fonctions sont de même nature : elles vont l'une comme l'autre construire une nouvelle matrice à partir de M passée en paramètre. Ainsi, chaque ligne M_i de M est transformée en zéro, une ou plusieurs lignes selon la forme de son premier élément $M_{i,1}$:

Pour chaque $i \in [1..m]$ *faire*

$M_{i,1}$	`spec`(C, M)	`default`(M)
attrape-tout / variable	$\__1 \cdots \__{\lvert C\rvert}$ `tail`$_i$	`tail`$_i$
$C'(a_1, \ldots, a_{\lvert C'\rvert})$	$\begin{cases} a_1 \cdots a_{\lvert C\rvert}$ `tail`$_i$ & \text{si } $C = C'$ \\ \text{pas de ligne} & \text{sinon} \end{cases}$	pas de ligne
$m_1 \mid m_2$	`spec` $\left(C, \begin{pmatrix} m_1 & \texttt{tail}_i \\ m_2 & \texttt{tail}_i \end{pmatrix}\right)$	`default` $\left(\begin{pmatrix} m_1 & \texttt{tail}_i \\ m_2 & \texttt{tail}_i \end{pmatrix}\right)$
m *as* x	`spec`$(C, m$ `tail`$_i)$	`default`$(m$ `tail`$_i)$
$c_1..c_2 \quad (c_1 \leq C \leq c_2)$	`tail`$_i$	pas de ligne

Soit $M_{m\times n}$ la matrice des motifs, et $\vec{q}_n$ le motif testé. L'algorithme `usefulness`$(M, \vec{q})$ est défini par :

Cas de base Si M n'a plus de colonnes (*i.e.* $n = 0$) alors le résultat dépend du nombre de lignes. Le vecteur $\vec{q}$ est bien filtré si et seulement si $m \neq 0$.

$$\texttt{usefulness}(M_{m\times 0}, \vec{q}) = \begin{cases} \mathit{vrai} & \text{si } m = 0 \\ \mathit{faux} & \text{sinon} \end{cases}$$

Induction Notons $\vec{q} = (q_1, \ldots, q_n)$.

1. Si q_1 est une valeur construite, *e.g.*, $q_1 = C(a_1, \ldots, a_k)$, alors la nouvelle matrice est construite ligne à ligne par la fonction $\texttt{spec}(C, M)$. Nous ferons l'appel récursif suivant :

$$\texttt{usefulness}(\texttt{spec}(C, M), \vec{q})$$

2. Si q_1 est une variable $q_1 = x$ ou un attrape-tout $q_1 = \_$, soit $\Sigma = \{C_1, C_2, \ldots, C_k\}$ l'ensemble des constructeurs apparaissant sur la première colonne de M (dans la matrice d'exemple $\Sigma = \{\texttt{Some}\}$). Nous dirons que cet ensemble est complet s'il contient tous les constructeurs du type de l'élément filtré. Raisonnons par cas en fonction de cette propriété :

   (a) Σ est complet, donc le constructeur de la future valeur filtrée est nécessairement un élément de Σ. Nous nous retrouvons alors dans le cas précédent, à ceci près que nous ne connaissons pas le constructeur filtré. Il faut donc tous les garder. Ainsi, le résultat renvoyé est :

$$\bigvee_{i=1}^{k} \texttt{usefulness}(\texttt{spec}(C_i, M), \texttt{spec}(C_i, \vec{q}))$$

   (b) Σ n'est pas complet (certains constructeurs du type courant n'apparaissent pas sur la première colonne), nous pouvons donc traiter uniquement les lignes commençant par un attrape-tout (ou une variable) en utilisant la seconde fonction définie plus haut. L'appel récursif sera donc :

$$\texttt{usefulness}(\texttt{default}(M), (q_2 \cdots q_n))$$

3. Si q_1 est un motif-ou, *i.e.*, $q_1 = q_{1.1} \,|\, q_{1.2}$, alors il faut simplement renvoyer la disjonction des deux sous-cas :

$$\texttt{usefulness}(M, (q_{1.1}, \ldots, q_n)) \vee \texttt{usefulness}(M, (q_{1.2}, \ldots, q_n))$$

2.4 Exemple

Nous allons dérouler notre algorithme à la main sur l'exemple suivant :

```
type t = A of t | B of int | C
let f = function
  | A (A t) -> 1
  | A (B _) -> 2
  | A C     -> 3
  | B x     -> 4
  | C       -> 5
```

$$M_{5,1} = \begin{pmatrix} A(A\ t) \\ A(B\ \_) \\ A\ C \\ B\ x \\ C \end{pmatrix}$$

On appelle donc $\texttt{usefulness}(M, \_)$. Comme $q_1 = \_$ et Σ est complet, on renvoie la disjonction des trois cas suivant :

1. $\texttt{usefulness}(\begin{pmatrix} A\ t \\ B\ \_ \\ C \end{pmatrix}, \_)$
   Dans ce premier cas, $q_1 = \_$ et Σ est complet, nous avons donc trois appels récursifs :

1.1 `usefulness`$((t), \_)$

Le premier élément de $\vec{q}$ est toujours un attrape-tout mais Σ n'est pas complet, fabriquons donc la matrice par défaut et faisons l'appel récursif correspondant :

1.1.1 `usefulness`$(\begin{pmatrix}\end{pmatrix}, \_)$

Ici notre matrice $M_{m\times n}$ n'a plus de colonne : $n = 0$ mais $m \neq 0$ donc nous renvoyons faux.

1.2 `usefulness`$((\_), \_)$

Comme les attrape-tout se comportent de façon strictement identique à des variables, ce cas se ramène à l'appel 1.1. Après l'appel à `usefulness`$(\begin{pmatrix}\end{pmatrix}, \_)$ nous renvoyons faux.

1.3 `usefulness`$(\begin{pmatrix}\end{pmatrix}, \_)$

Nous sommes dans le cas de base où la matrice n'a plus de colonnes ($n = 0$) mais $m \neq 0$, donc le résultat est faux.

2. `usefulness`$((x), \_)$

Comme pour le cas 1.1, le résultat est faux.

3. `usefulness`$(\begin{pmatrix}\end{pmatrix}, \_)$

Comme pour le cas 1.3, le résultat est faux.

La disjonction des résultats des trois appels récursifs est donc fausse ce qui indique — comme nous l'attendions — que notre filtrage est exhaustif.

2.5 Terminaison

Définition 13 (Poids d'une matrice)**.** Soit $M \in \mathcal{M}_{m\times n}(\mathbb{M})$ une matrice de filtrage. On défini le poids de M, noté *poids*(M), par le triplet $\langle \#(\text{motifs-ou}), \textit{taille}(M), n\rangle$. La première composante est le nombre total de motifs-ou apparaissant dans la M. Ce triplet permet de définir une relation d'ordre lexicographique sur les matrices de filtrage, notée $\preceq$.

Remarque 4. Nous utiliserons plus particulièrement la relation stricte associée, que nous noterons $\prec$. Celle-ci est bien fondée car il s'agit de l'ordre lexicographique du produit de trois ordres bien fondés (ordre usuel sur les entiers naturels).

L'algorithme `usefulness`$(M_{m\times n}, \vec{q})$ étant récursif, il suffit pour prouver la terminaison de montrer que le poids décroît strictement à chaque appel récursif.

Théorème 1. *Le variant de* `usefulness` *est poids*(M) *: chaque appel récursif est réalisé sur une matrice de poids plus faible que la matrice courante.*

Démonstration. Raisonnons par cas en suivant la structure de l'algorithme :

Induction Un appel récursif peut être construit de deux façons, avec la matrice spécifique à un constructeur C obtenu par la fonction `spec`(C, M), ou avec la matrice par défaut `default`(M). Montrons que dans les deux cas, pour la matrice M' ainsi construite nous aurons *poids*$(M') \prec$ *poids*(M).

Supposons donc que l'on ait :

— `type t = C`$_0$ `| C`$_1$ `of p`$_1$ `| C`$_a$ `of p`$_1$ `* ⋯ * p`$_a$,

— $M : \tau$

— $\tau =$ `(t, t, t)`

Soit $\langle x, y, z\rangle$ le poids de M, raisonnons par cas sur la première colonne de la matrice M.

— Si Σ est complet, M peut être de la forme proposée en figure (1). Dans cet exemple, remarquons que le motif-ou qui apparaissait en tête de la ligne 3 disparaît, donc le nombre global de x diminue. La seconde coordonnée y ne peut que diminuer, sauf s'il y avait un motif-ou avec un constructeur

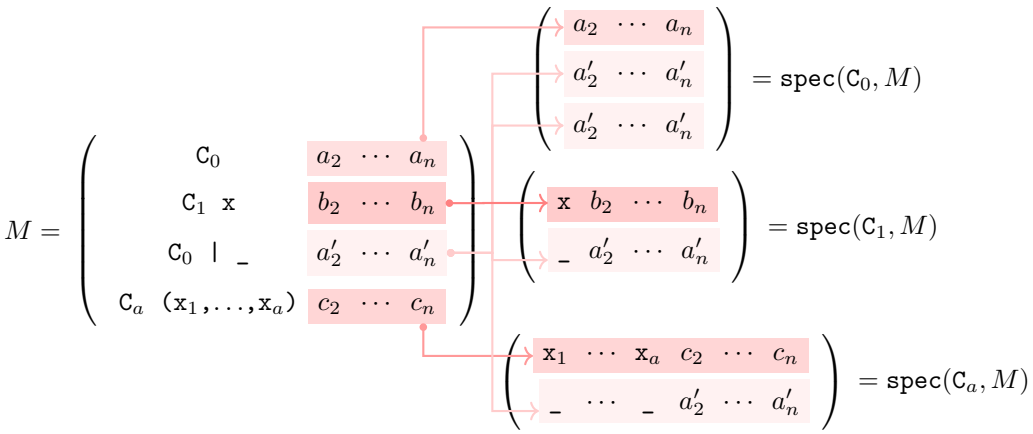

FIGURE 1 – Forme de la matrice de filtrage lorsque l'ensemble Σ est complet.

de notre type d'un coté, ce qui engendre une duplication de ligne comme pour $\texttt{spec}(\texttt{C}_0, M)$. Enfin, le nombre de colonnes augmente de $a-1$ pour la matrice spécialisée d'un constructeur d'arité a (un constructeur d'arité 0 va faire diminuer d'exactement 1 le nombre de colonnes).
De manière générale, si un motif-ou apparaît en première colonne, celui-ci va disparaître ce qui engendre une diminution de la coordonnée x de notre variant. Sinon lorsqu'aucun motif-ou n'apparaît en première colonne, nous y retrouvons nécessairement l'ensemble complet des constructeurs donc les matrices spécialisées sont fabriquées. Chacune d'entre elles va être de poids plus faible que M car :

— la (ou les) ligne(s) correspondantes au constructeur en cours de spécialisation sont de taille inférieure à leur taille initiale car le constructeur en question disparaît ;
— les lignes commençant par des variables/attrape-tout sont gardées mais n'augmentent pas la taille, car même si l'on duplique des attrape-tout, on ne fait qu'ajouter 0 plusieurs fois ;
— les autres lignes disparaissent.

Ainsi, nous avons bien

$$\forall i.\ (i \in \{0, 1, n\}) \Rightarrow \mathit{poids}(\texttt{spec}(\texttt{C}_i, M)) \prec \mathit{poids}(M)$$

— Sinon, si Σ n'est pas complet nous avons deux cas :

1. La première colonne de M ne contient pas d'attrape-tout ou de variable. Dans ce cas, la matrice par défaut construite est vide et nous avons donc bien

$$\mathit{poids}(\texttt{default}(M)) = e \prec \mathit{poids}(M)$$

2. La première colonne de M contient au moins un attrape-tout (ou une variable), possiblement sous un motif-ou, M peut être de la forme proposée en figure (2). Comme pour le cas `spec`, l'ensemble des motifs-ou en tête disparaissent. Si des constructeurs sont présents, la taille diminue d'au moins 1 pour chacun d'entre eux. Enfin, si M ne contient que des attrape-tout et/ou variables, alors le nombre de colonnes diminue. Ainsi,

$$\forall M.\ (\mathit{poids}(M) \neq e) \Rightarrow \mathit{poids}(\texttt{default}(M)) \prec \mathit{poids}(M)$$

Conclusion Nous constatons donc bien que, quel que soit le cas, un appel récursif est par construction réalisé sur une matrice de poids strictement plus faible que la matrice courante. Cette dernière ne fait donc que décroître, jusqu'à atteindre le cas de base de l'algorithme `usefulness`. □

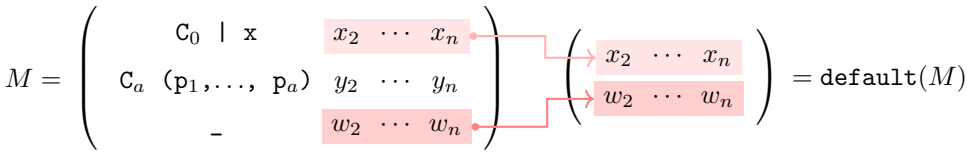

FIGURE 2 – Forme de la matrice de filtrage lorsque l'ensemble Σ est incomplet et que la première colonne contient un attrape-tout.

Corollaire 1. *L'algorithme* ***usefulness*** *termine.*

2.6 Correction

La preuve de la correction partielle de `usefulness` nécessite les deux lemmes suivants.

Lemme 1. *La relation de filtrage est préservée par spécialisation.*

$$\forall M \in \mathcal{M}_{m\times n}(\mathbb{M}) : \tau. \forall \vec{q} \in \mathbb{M}^n : \tau. \vec{q} = (C(a_1, \ldots, a_k)\ q_2 \cdots q_n) \Rightarrow (M \preceq \vec{q} \Leftrightarrow \textbf{\textit{spec}}(C, M) \preceq \textbf{\textit{spec}}(C, \vec{q}))$$

Démonstration. Soient $\vec{q} = (C(a_1, \ldots, a_k)\ q_2 \cdots q_n) : \tau$ et $M \in \mathcal{M}_{m\times n}(\mathbb{M}) : \tau$.

$$\begin{aligned}
M \preceq \vec{q} &\Leftrightarrow \exists i \in [1..m].\ \forall j \in [1..n].\ M_{i,j} \preceq q_j \\
&\Leftrightarrow \exists i \in [1..m].\ M_{i,1} = C(a_1^M, \ldots, a_k^M) \preceq q_1 \wedge \mathtt{tail}_i \preceq q_2 \cdots q_n \\
&\Leftrightarrow \forall j \in [1..k].\ a_j^M \preceq a_j \wedge \mathtt{tail}_i \preceq q_2 \cdots q_n \\
&\Leftrightarrow a_1^M \cdots a_k^M\ \mathtt{tail}_i \preceq a_1 \cdots a_k\ q_2 \cdots q_n \\
&\Leftrightarrow \mathtt{spec}(C, M) \preceq \mathtt{spec}(C, \vec{q})
\end{aligned}$$

□

Lemme 2. *Pour toute matrice de filtrage M et vecteur de motif $\vec{q}$ de même type, si M filtre $\vec{q}$ alors la matrice par défaut de M filtre la matrice par défaut de $\vec{q}$:*

$$\forall M \in \mathcal{M}_{m\times n}(\mathbb{M}) : \tau. \forall \vec{q} \in \mathbb{M}^n : \tau.\ M \preceq \vec{q} \Rightarrow \textbf{\textit{default}}(M) \preceq \textbf{\textit{default}}(\vec{q})$$

Démonstration. Soient $M \in \mathcal{M}_{m\times n}(\mathbb{M}) : \tau$ et $\vec{q} = (q_1 \cdots q_n) : \tau$.

$$\begin{aligned}
M \preceq \vec{q} &\Rightarrow \exists i \in [1..m].\ \forall j \in [1..n].\ M_{i,j} \preceq q_j \\
&\Rightarrow \exists i \in [1..m].\ M_{i,1} \preceq q_1 \wedge \mathtt{tail}_i \preceq q_2 \cdots q_n \\
&\Rightarrow \mathtt{tail}_i \preceq q_2 \cdots q_n \\
&\Rightarrow \mathtt{default}(M) \preceq \mathtt{default}(\vec{q})
\end{aligned}$$

□

Théorème 2. *Le résultat de* ***usefulness***$(M, \vec{q})$ *est vrai si et seulement si aucune ligne de M ne filtre $\vec{q}$.*

Démonstration. Prouver ce théorème revient à montrer que `usefulness`$(M_{m\times n}, \vec{q})$ satisfait le couple ⟨ précondition, postconditions ⟩, avec « *M et $\vec{q}$ sont de même type* » comme précondiction et « *le résultat est vrai si M n'a plus de lignes* » comme postcondition dans le cas de base ($n = 0$) et enfin la postconditions dans le cas inductif ($n > 0$) est « *Si M filtre q alors l'un des appels récursifs est réalisé sur M' et q' tels que M' filtre q'.* ».

Raisonnons sur le code de `usefulness`.

Cas de base La postcondiction correspond précisément à l'algorithme.

Induction Notons $\vec{q} = (q_1, \ldots, q_n)$.

1. Soit q_1 est une valeur construite, de constructeur C. Nous fabriquons la matrice spécialisée pour C. Le lemme (1) nous assure donc que $\texttt{spec}(C, M)$ filtre $\texttt{spec}(C, \vec{q})$ si et seulement si M filtre q, ce qui conclut ce point.
2. Soit q_1 est une variable ou un attrape-tout.
   (a) Σ est complet (de taille k), nous construisons l'ensemble des k matrices spécialiséees. Nous savons que si M filtre $\vec{q}$ alors exactement une des matrices $\texttt{spec}(C_i, M)$ filtre $\texttt{spec}(C_i, \vec{q})$ (lemme (1)) ce i-ème cas fait bien partie des k appels récursifs, la postcondition est bien vérifiée.
   (b) Σ n'est pas complet, nous construisons la matrice par défaut et le lemme (2) conclut ce cas.
3. Soit q_1 est un motif-ou $(q_1 = q_{1.1} \mid q_{1.2})$, la postcondition inductive, est trivialement vérifiée. Soient $\vec{q_l} = (q_{1.1}, \ldots, q_n)$ et $\vec{q_r} = (q_{1.2}, \ldots, q_n)$, si M filtre $\vec{q}$, alors M filtre $\vec{q_l}$ et $\vec{q_r}$ (cf. règle OR_0 dans la définition de la relation de filtrage).

□

Conclusion Nous avons démontré la correction partielle ainsi que la terminaison, donc nous avons prouvé la correction totale de `usefulness`.

2.7 Idée de complexité

L'algorithme de clause utile est NP-complet (Ron Sekar *et al.*, 1992 [20]). Nous pouvons assez simplement construire une instance qui engendrera un nombre exponentiel d'appels récursifs. Dans un premier temps, prenons une matrice triangulaire supérieure à coefficients dans $\mathbb{M}$ dont les valeurs sous la diagonale sont des `_` et les autres des `A` (par exemple, cette matrice en dimension 2 est : $\left(\begin{smallmatrix} \texttt{A} & \texttt{A} \\ \texttt{\_} & \texttt{A} \end{smallmatrix}\right)$). Ensuite, dupliquons une à une chaque ligne et remplaçons sur celle-ci les `A` par des `B`. Enfin, nommons f_n la matrice de taille $2n \times n$ construite par cette procédure.

```
type t = A | B
let f4 = function
| A,A,A,A  -> 0
| B,B,B,B  -> 1
| _,A,A,A  -> 2
| _,B,B,B  -> 3
| _,_,A,A  -> 4
| _,_,B,B  -> 5
| _,_,_,A  -> 6
| _,_,_,B  -> 7
```

FIGURE 3 – Équivalent de la matrice f_4 en OCaml

Remarque 5. En appliquant l'algorithme `usefulness` naïvement sur la matrice f_4, nous nous retrouvons à faire 32 appels récursifs. Pour imaginer cette croissance, il suffit de comprendre que $\texttt{usefulness}(f_4, \texttt{\_})$ va en quelque sorte appeler $\texttt{usefulness}(f_3, \texttt{\_})$ deux fois, et récursivement jusqu'à f_0. Ainsi, nous nous retrouvons avec un arbre binaire de hauteur 5, ce qui correspond bien à nos 32 appels observés. La figure (4) illustre la trace du premier appel en profondeur réalisé jusqu'à f_0 (il s'agit de la première branche complète développée de l'arbre).

Notation 3 (Complexité en temps)**.** Soit $\mathcal{A}$ un algorithme et x son entrée. Nous noterons $\mathcal{C}_{\text{TIME}}(\mathcal{A}(x))$ la complexité en temps de l'exécution de l'algorithme $\mathcal{A}$ sur l'entrée x.

Théorème 3 (Complexité exponentielle)**.** *Pour tout $n > 0$, la complexité de* $\texttt{usefulness}(f_n, (\texttt{\_} \cdots \texttt{\_}))$ *est exponentielle.*

Pour démontrer le théorème (3) nous avons besoin du lemme suivant.

Lemme 3 (Non simplification par duplication de ligne)**.** *Pour toute matrice de filtrage M, la présence de d duplications de la première ligne dans M ne peut pas améliorer la complexité de l'algorithme* `usefulness`*. Au contraire, si l'on duplique d fois la première ligne de M, la complexité sur la nouvelle matrice sera au moins aussi coûteuse :*

$$\mathcal{C}_{\text{Time}}(\texttt{usefulness}(M, \_ \cdots \_)) \leq \mathcal{C}_{\text{Time}}((\texttt{usefulness}(M_{duplicated}, \_ \cdots \_)))$$

Démonstration. Soit $M \in \mathcal{M}_{m\times n}(\mathbb{M})$ une matrice de filtrage. Notons M_d une copie de M dans laquelle la première ligne a été dupliquée d fois, avec $d \geq 0$. Remarquons que M_d est de dimension $(m+d) \times n$ et que $M = M_0$. Montrons que la complexité de `usefulness` sur M_d est au mieux égale, sinon pire que sur la matrice M. Soit $\mathcal{P}_d$ la proposition « $\mathcal{C}_{\text{Time}}(\texttt{usefulness}(M, \_ \cdots \_)) \leq \mathcal{C}_{\text{Time}}(\texttt{usefulness}(M_d, \_ \cdots \_))$ ». Montrons P_d par récurrence sur le code de `usefulness` (cette récurrence est bien fondée, nous avons en effet prouvé en partie 2.5 que `usefulness` termine).

Cas de base Si M n'a plus de colonnes (*i.e.* $n = 0$) alors le résultat dépend du nombre de lignes. Nous avons 2 cas, $m = 0$ et $m \neq 0$, or la nullité de m est indépendante de la valeur de d. Ainsi, quelle que soit la valeur de d, la complexité en temps est égale, donc $\mathcal{P}_d$ est vrai, pour le cas de base.

Induction Comme pour la définition de `usefulness`, notons $\vec{q} = (q_1, \ldots, q_n)$. Ici, nous savons déjà que q_1 est un attrape-tout, nous n'avons donc qu'un seul cas parmi les trois cas de l'algorithme à traiter.

Deux sous-cas apparaissent :

1. L'ensemble des constructeurs apparaît sur la première colonne. Nous allons donc construire pour chacun une matrice spécialisée. Cette construction est réalisée en temps linéaire pour M : il suffit de parcourir la matrice et de sélectionner les lignes commençant par le constructeur correspondant. Le temps est donc proportionnel au nombre de lignes présentes dans la matrice. D'autre part, pour la matrice M_d, la complexité reste linéaire mais celle-ci est affectée par les d lignes supplémentaires. Nous passons d'un facteur de m (nombre de lignes de M) à un facteur de $m + d$ (nombre de lignes de M_d). Nous observons donc une égalité asymptotique, mais l'inégalité $\mathcal{P}_d$ est vérifiée.
2. Il manque des constructeurs sur la première colonne. Dans ce cas la matrice par défaut est construite. Comme pour le cas précédent, cette construction est réalisée en temps linéaire pour M ainsi que pour M_d, il suffit de parcourir la matrice et de sélectionner les lignes commençant par un attrape-tout ou une variable. Le temps est proportionnel au nombre de lignes présentent dans la matrice, ainsi nous avons un surcoût lors du traitement de M_d. L'inégalité $\mathcal{P}_d$ est donc bien vérifiée.

Conclusion Ainsi par récurrence bien fondée sur le code de `usefulness`, nous avons prouvé que pour tout $d \geq 0$, la propriété $\mathcal{P}_d$ est vrai. □

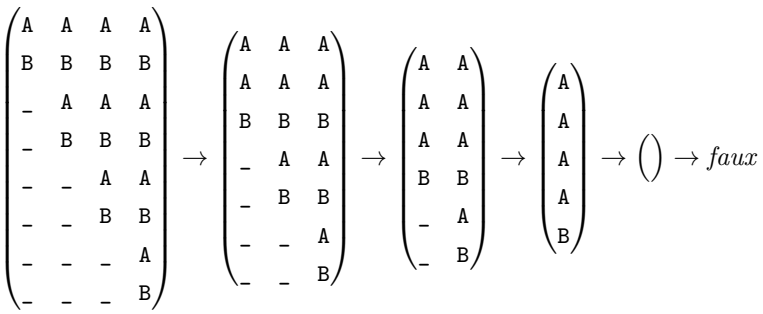

FIGURE 4 – Trace partielle du premier appel récursif en profondeur de `usefulness`$(f_4, \_ \cdots \_)$.

Nous pouvons à partir de ce lemme prouver que le comportement de l'algorithme `usefulness` est bien exponentiel sur tout entrée de la forme f_n.

Démonstration. L'équation à résoudre est la suivante [4] : $C(n) \geq 2C(n-1) + 4n$. En effet, comme nous avons seulement 2 constructeurs et que chaque colonne est complète, chaque appel récursif sera fait avec les matrices $\texttt{spec}(\texttt{A}, M)$ et $\texttt{spec}(\texttt{B}, M)$ et chacune d'entre elle sera constituée d'exactement $n-1$ colonnes car $\texttt{spec}(\texttt{A}, M)$ supprime uniquement la première composante comme il s'agit de constructeurs d'arité 0 (nous avons le même comportement pour $\texttt{spec}(\texttt{B}, M)$ qui ne supprime uniquement que la première colonne aussi). Le facteur linéaire $4n$ supplémentaire correspond aux deux traversées de M pour la construction des sous-matrices spécifiques : M a $2n$ lignes et nous faisons 2 fois le parcours. Enfin, remarquons que par construction $\vec{q}$ sera toujours un vecteur d'attrape-tout.

Résolution de l'équation récursive :

$$\begin{aligned} C(n+1) &\geq 2C(n) + 4(n+1) \\ \frac{C(n+1)}{2^{n+1}} &\geq \frac{2C(n) + 4(n+1)}{2^{n+1}} \\ \frac{C(n+1)}{2^{n+1}} &\geq \frac{C(n)}{2^n} + \frac{n+1}{2^{n-1}} \end{aligned}$$

On pose $U(n) = \frac{C(n)}{2^n}$:

$$\begin{aligned} U(n+1) &\geq U(n) + \frac{n+1}{2^{n-1}} \\ U(n+1) &\geq \sum_{i=0}^{n} \frac{i+1}{2^{i-1}} \end{aligned}$$

Pour $n \geq 1$:

$$\begin{aligned} U(n+1) &\geq 2 \\ \frac{C(n+1)}{2^{n+1}} &\geq 2 \end{aligned}$$

D'où,

$$C(n+1) \geq 2^{n+2}$$

Conclusion Cette inégalité conclut la preuve de la possible explosion exponentielle de l'algorithme `usefulness`.

□

Remarque 6 (Choix de la colonne). L'exemple précédent illustre bien l'impact que peut avoir l'ordre dans lequel les colonnes sont traitées. En effet, en les prenant de gauche à droite, nous nous retrouvons à faire 32 appels récursifs, alors que si nous avions traité en premier la dernière colonne nous n'en aurions eu que 10. Ainsi, il semble que certaines configurations de matrice sont significativement meilleures que d'autres.

4. Le problème au rang $n-1$ n'est pas exactement le même que celui du rang n (cf. trace partielle d'exécution illustrée en figure (4)). Si cela avait été le même problème au rang $n-1$ nous aurions eu une égalité et non une inégalité. En effet, lors des appels récursifs sur les sous-matrices nous avons un phénomène de duplication de lignes en tête. Ainsi, grâce au lemme (3) nous savons que si la première ligne est dupliquée un nombre arbitraire de fois, alors la complexité est au mieux égale.

2.8 Génération de contre-exemples

L'information de la non exhaustivité est un atout non négligeable. Cependant, un bit d'information n'est pas toujours suffisant. Nous pouvons tout à fait imaginer un filtrage profond sur un type algébrique complexe dans lequel il serait pénible de déceler sans aucune aide le motif manquant. Il est ainsi raisonnable de chercher à étendre l'algorithme `usefulness` afin de ne plus seulement indiquer l'utilité, mais aussi fournir un contre-exemple en cas de non exhaustivité. C'est un travail qui est par exemple réalisé par le compilateur OCaml, qui en cas de filtrage partiel nous indique :

```
let f = function      Warning 8 [partial-match]: this pattern-matching is not exhaustive.
    | []   -> 0       Here is an example of a case that is not matched:
    | 1::t -> 1       0::_
```

Cet algorithme est décrit dans le même article (Maranget 2007 [13]). Il s'agit d'une spécialisation de `usefulness`, visant à produire le contre-exemple de motif souhaité. Appellons $\mathtt{U}_{\mathtt{CE}}$, pour `usefulness` avec contre-exemple, notre extension de `usefulness`. N'ayant plus le besoin de tester un motif nous ne prenons que la matrice M en paramètre, en vue de renvoyer un contre-exemple s'il existe, et $\bot$ sinon. L'algorithme $\mathtt{U}_{\mathtt{CE}}$ est défini par :

Cas de base Si M n'a plus de colonnes (*i.e.* $n = 0$) alors comme pour `usefulness`, il y a deux cas de base en fonction du nombre m de lignes :

$$\mathtt{U}_{\mathtt{CE}}(M_{m\times 0}) = \begin{cases} () & \text{si } m = 0 \\ \bot & \text{sinon} \end{cases}$$

Induction

1. Soit $\Sigma = \{C_1, C_2, \ldots, C_k\}$ l'ensemble des constructeurs apparaissant sur la première ligne de M. Par cas sur la complétude de Σ :
   (a) Si Σ est complet, faisons pour tout $C_i \in \Sigma$ l'appel $\mathtt{U}_{\mathtt{CE}}(\mathtt{spec}(C_i, M))$. Observons deux cas :
   — dès que l'un de ces appels rend un vecteur de motifs $(a_1 \cdots a_{|C_k|} \;\; m_2 \cdots m_n)$, le résultat est ce même vecteur dans lequel les $|C_k|$ premiers éléments seront réunis sous le constructeur C_k, c'est-à-dire le vecteur $(C_k(a_1 \cdots a_{|C_k|}) \;\; m_2 \cdots m_n)$.
   — l'ensemble de ces appels renvoient la valeur $\bot$ alors le résultat de $\mathtt{U}_{\mathtt{CE}}(M)$ est aussi $\bot$.
   (b) Si Σ n'est pas complet, nous devons seulement faire un appel récursif sur la matrice par défaut de M. Comme pour le cas précédent, si $\mathtt{U}_{\mathtt{CE}}(\mathtt{default}(M)) = \bot$ alors $\mathtt{U}_{\mathtt{CE}}(M) = \bot$. Sinon si $\mathtt{U}_{\mathtt{CE}}(\mathtt{default}(M)) = (m_2 \cdots m_n)$ alors le résultat est dépendant de Σ :

$$\mathtt{U}_{\mathtt{CE}}(M) = \begin{cases} (\_ \; m_2 \cdots m_n) & \text{si } \Sigma = \emptyset \\ (C(\__1 \cdots \__{|C_k|}) \; m_2 \cdots m_n) & \text{sinon, avec } C \notin \Sigma \end{cases}$$

2.9 Implémentation

Nous avons donc implémenté cet algorithme `usefulness` à l'intérieur du code de GOSPEL. Un appel à cette fonction est donc fait au cours du typage de la spécification logique pour chaque endroit où se trouve un filtrage. Nous avons décidé de lever une erreur lorsqu'une construction de filtrage non exhaustive est détectée. En revanche, la présence de redondance ne fait que déclencher un avertissement. L'ensemble du code peut être retrouvé sur la page GitHub du projet GOSPEL [5].

5. https://www.github.com/ocaml-gospel/gospel

2.9.1 Tests

Pour vérifier l'absence de bogues dans l'implémentation nous avons conçu `minipat`, un langage exclusivement dédié au filtrage. Ceci nous a permis de tester notre implémentation contre celle d'OCaml en tant qu'implémentation de référence sur des problèmes de filtrage arbitrairement longs. Ce langage étant compatible avec OCaml, il suffit de trouver une façon de générer des motifs afin de comparer les résultats donnés par notre algorithme et le compilateur OCaml, pour nous indiquer d'éventuelles incohérences.

Générateur de fichiers types Pour la génération de fichiers compilables, deux étapes sont requises (la troisième étant optionnelle) :

1. génération d'un type algébrique ;
2. génération de motifs formant un filtrage exhaustif relativement au type défini en (1) ;
3. *optionnel : casser l'exhaustivité pour avoir des tests négatifs.*

Génération aléatoire d'un type algébrique Un type OCaml `t` peut être défini par une liste d'association « constructeur → arguments ». Leur génération aléatoire est relativement simple. Il suffit de fixer le nombre de constructeurs (donner à chacun un identifiant différent, par exemple prendre les lettres l'alphabet dans l'ordre), puis de tirer aléatoirement l'arité de chacun. Ensuite, choisir au hasard le type de chaque argument dudit constructeur parmi un ensemble $\mathcal{T}$ défini en amont. Cet ensemble peut par exemple valoir [6] :

$$\mathcal{T}_{\texttt{t}} \;=\; \{\texttt{int},\ \texttt{string},\ \texttt{t}\}$$

Génération aléatoire d'un filtrage exhaustif pour un type donné Soit `t` un type, tel que :

$$\texttt{type t} = C_1\ \texttt{of}\ \texttt{t}_{1,1} \cdots \texttt{t}_{1,|C_1|} \mid \cdots \mid C_n\ \texttt{of}\ \texttt{t}_{n,1} \cdots \texttt{t}_{n,|C_n|}$$

Nous pouvons fabriquer un filtrage OCaml exhaustif avec l'algorithme $\mathcal{F}_{\text{alea}}$. Ce dernier a trois paramètres : deux entiers `max_depth` et `max_rows`, et le type `t` du filtrage que l'on veut construire. Le retour de $\mathcal{F}_{\text{alea}}$ est une liste de motifs formant un filtrage exhaustif et bien typé pour `t`. Ce filtrage contient au plus `max_rows` lignes et chaque motif a une profondeur au plus `max_depth`. L'algorithme $\mathcal{F}_{\text{alea}}$ est defini récursivement de la manière suivante :

Cas de base (1) Si `t` est un type de base (i.e. `int`, `string`, `char`), soit k un entier tiré au hasard entre 1 et `max_rows` $- 1$, construire k valeurs différentes du type `t` et renvoyer cette liste, suivie d'un attrape-tout. Le résultat est donc `[`$v_1, \ldots, v_k$`, _]`, où tout v_i est de type `t`.

Cas de base (2) Si `max_depth` $= 0$, renvoyer la liste singleton ne contenant qu'un attrape-tout, notée `[_]`.

Cas de base (3) Si `max_rows` $= 1$, renvoyer `[_]`.

6. Où `t` est le nom du type lui-même, ce qui permettra d'introduire de la profondeur dans les motifs générés.

Induction Nécessairement, `t` est un type algébrique (défini par l'utilisateur).

— Si $|\texttt{t}| > \texttt{max\_rows}$ alors renvoyer `[_]`.

— Sinon $|\texttt{t}| \leq \texttt{max\_rows}$, soit $\texttt{k} = \lfloor \frac{\texttt{max\_rows}}{|\texttt{t}|} \rfloor$. Pour tout C_i, générer une liste exhaustive de motifs pour chacun de ses j arguments (pour $j \in [1..|C_i|]$) avec l'appel $\mathcal{F}_{\text{alea}}$`(max_depth - 1, k, t`$_{i,j}$`)`. Une fois ces $|C_i|$ appels réalisés, faire le produit cartésien des listes obtenues et leur appliquer le constructeur C_i. Si la taille de ce produit cartésien dépasse k, ne conserver que $k-1$ des motifs et ajouter un motif C_i`(_,...,_)`. Nous avons donc généré pour chaque C_i un nombre de lignes inférieur ou égale à k. Enfin, renvoyer la liste des i listes concaténées.

Remarque 7. Pour ajouter de l'aléa dans cette construction, nous pouvons :

— faire un mélange (par exemple Knuth [10]) sur la liste une fois construite ;

— sélectionner certaines des lignes aléatoirement pour en faire des motifs-ou ;

— ajouter un quatrième cas de base qui se déclenche de manière aléatoire (par exemple avec probabilité $\frac{1}{4}$) et qui comme les autres renvoie `[_]` mais sans condition.

Exemple d'exécution Illustrons le fonctionnement de cet algorithme en déroulant un exemple d'exécution à la main. Supposons que l'on fasse l'appel :

$\mathcal{F}_{\text{alea}}$`(max_depth = 1, max_rows = 10, type t = A of int * t * char | B of int)`

On rentre directement dans le *sinon* du cas inductif, nous avons $\texttt{k} = \frac{10}{2} = 5$.

— Pour `A` :

— Pour `int`, nous faisons l'appel $\mathcal{F}_{\text{alea}}$`(0, 5, int)`. Nous rentrons dans le cas de base (1), supposons $k = 2$, nous avons deux entiers à construire, nous renvoyons donc par exemple la liste `[1,0,_]`.

— Pour `t`, nous faisons l'appel $\mathcal{F}_{\text{alea}}$`(0, 5, t)`. Nous rentrons dans le cas de base (2) donc le résultat est `[_]`.

— Pour `char`, nous faisons l'appel $\mathcal{F}_{\text{alea}}$`(0, 5, char)`. Nous rentrons dans le cas de base (1), supposons $k = 2$, nous avons deux caractères à construire, nous renvoyons donc par exemple la liste `['p','a',_]`.

Après les trois sous-appels récursifs nous récupérons trois listes dont nous devons faire le produit cartésien pour fabriquer les lignes à renvoyer (chacune sous le constructeur `A`), ce qui nous donne :

```
[ [A(1,_,'p')]; [A(1,_,'a')]; [A(1,_,_)];
  [A(0,_,'p')]; [A(0,_,'a')]; [A(0,_,_)];
  [A(_,_,'p')]; [A(_,_,'a')]; [A(_,_,_)] ]
```

Comme nous ne voulons que 5 lignes, nous gardons les quatre premières et la dernière, ce qui nous donne :

```
[ [A(1,_,'p')]; [A(1,_,'a')]; [A(1,_,_)];
  [A(0,_,'p')]; [A(_,_,_)] ]
```

— Pour `B` :

— Pour `int`, nous faisons l'appel $\mathcal{F}_{\text{alea}}$`(0, 10, int)`. Nous rentrons dans le cas de base (1), supposons $k = 1$, nous avons un entier à construire, nous renvoyons donc par exemple la liste `[5,_]`.

Nous mettons simplement ce résultat sous le constructeur `B` et on le renvoie : `[B 5; B _]`.

Enfin, la concaténation résultante est :

```
[ [A(1,_,'p')]; [A(1,_,'a')]; [A(1,_,_)];
  [A(0,_,'p')]; [A(_,_,_)]; [B 5]; [B _] ]
```

Remarquons que cette liste de motifs forme bien un filtrage exhaustif pour le type `t`.

Cassage de l'exhaustivité Pour casser l'exhaustivité d'une liste de motifs, il suffit de supprimer quelques lignes bien choisies.

Automatisation de la vérification Le langage `minipat` étant compatible avec OCaml, nous pouvons comparer notre implémentation de `usefulness` contre le compilateur OCaml. En effet, la compilation d'un fichier contenant un filtrage non exhaustif avec la commande [7] `ocamlc -w -11-12 -warn-error +8`, échoue avec le code 2. Il suffit donc de programmer le même comportement dans `minipat` afin de vérifier la correspondance. Cette phase de tests a permis de trouver un certain nombre de bogues dans l'implémentation de l'algorithme `usefulness`. Générant des filtrages arbitrairement longs, larges et/ou profonds, nous avons pu vérifier que l'implémentation était correcte sur un grand nombre de tests [8]. Le code de ce générateur peut être retrouvé en suivant ce lien [9].

2.10 Clauses `when`

Dans le langage OCaml, la construction de filtrage par motifs admet l'ajout optionnel de clauses `when`. Ces clauses sont des gardes permettant d'ajouter une conditionnelle sur certaines lignes de notre filtrage [10]. Une ligne comprenant un `when` est dite *gardée*. Les gardes permettent d'exprimer des conditions que les motifs ne capturent pas, par exemple les deux fonctions suivantes sont strictement équivalentes mais nous pouvons trouver que la seconde est plus élégante :

```
let pair_eq = function
  | x, y -> if x = y then 1 else 0
  | _    -> assert false
```

```
let pair_eq_when = function
  | x, y when x = y -> 1
  | _               -> 0
```

2.10.1 Exhaustivité avec les clauses `when`

Les clauses `when` ajoutent un niveau de difficulté à notre vérification d'exhaustivité du filtrage. Pour résoudre ce problème, une simple transformation du code peut être réalisée en amont afin de se ramener à un cas compatible avec l'algorithme `usefulness`. Celle-ci est inspirée de la transformation proposée par John Reppy *et al.* [17]. Ainsi, pour chaque occurrence de `when` nous ajoutons une colonne remplie d'attrape-tout à la matrice sauf pour la ligne gardée correspondante pour laquelle il faut mettre le booléen `true`. Par exemple, le code OCaml suivant va donner la matrice :

```
let f = function
  | A x, 42, y when x = y     -> 1
  | B,    5, _                -> 2
  | C a,  b, c when a + c = y -> 3
```

$$\rightarrow \quad \begin{pmatrix} \texttt{A x} & \texttt{42} & \texttt{y} & \texttt{true} & \texttt{\_} \\ \texttt{B} & \texttt{5} & \texttt{\_} & \texttt{\_} & \texttt{\_} \\ \texttt{C a} & \texttt{b} & \texttt{c} & \texttt{\_} & \texttt{true} \end{pmatrix}$$

Théorème 4. *Soient $M_g \in \mathcal{M}_{m\times n}(\mathbb{M})$ une matrice de filtrage gardée (avec g lignes gardées), $\vec{w} = (w_1, \ldots, w_g)$ le vecteur des gardes (avec $g \leq n$) et $M_t \in \mathcal{M}_{m\times(n+g)}(\mathbb{M})$ la matrice dé-gardée à partir de M_g en suivant la procédure indiquée. Si M_t est exhaustive, alors M_g l'est aussi.*

Démonstration. Comme M_t est exhaustive, alors pour tout $\vec{q_t}$, M_t filtre $\vec{q_t}$. Montrons $\forall \vec{q_g}.\, M_g \preceq \vec{q_g}$. Soit $\vec{q_g}$ un vecteur de motifs. On construit un vecteur $\vec{q_t}$ en ajoutant à $\vec{q_g}$ les g booléens construits de la

8. Plus de 10 000 problèmes générés soit environ 500 000 lignes de filtrage.

9. https://www.github.com/paulpatault/minipat

9. L'option `-w` permet d'ajouter/supprimer des avertissements, les codes 11 et 12 correspondent à la redondance et le - indique leur suppression. De plus l'option `-warn-error` permet de transformer les avertissements en erreurs et 8 est le code de l'exhaustivité.

10. Attention, contrairement à d'autres langages ML, une seule clause `when` n'est autorisée par ligne, et celle-ci s'applique à tous les motifs de la ligne.

manière suivante. L'élément b_j est défini tel que si $(M_g)_j$ filtre $\vec{q_g}$ alors $b_j \leftarrow w_j$ et sinon $b_j \leftarrow$ `true`. Par hypothèse, M_t filtre $\vec{q_t}$ donc il existe une ligne i telle que $(M_t)_i$ filtre $\vec{q_t}$. Montrons que $(M_g)_i$ filtre $\vec{q_g}$. Si il y a une garde j sur la ligne i, alors le booléen b_j a la valeur `true` (par définition de la matrice M_t) et donc la garde w_j est vraie et la ligne i de M_g filtre bien $\vec{q_g}$. Sinon, il n'y a pas de garde sur la ligne i et $(M_t)_i$ filtre $\vec{q_t}$ signifie exactement que $(M_g)_i$ filtre $\vec{q_g}$ car les n premières colonnes de M_t et M_g sont identiques. □

Remarque 8. La réciproque du théorème (4) n'est en revanche pas vraie. En effet, prenons un simple exemple :

```
let f = function
  | x, y when x = y ->  0
  | x, y when x < y -> -1
  | x, y when x > y ->  1
```

$$\rightarrow \quad \begin{pmatrix} \texttt{x} & \texttt{y} & \texttt{true} & \texttt{\_} & \texttt{\_} \\ \texttt{x} & \texttt{y} & \texttt{\_} & \texttt{true} & \texttt{\_} \\ \texttt{x} & \texttt{y} & \texttt{\_} & \texttt{\_} & \texttt{true} \end{pmatrix}$$

Dans ce cas, l'application de l'algorithme `usefulness` nous informe que le filtrage n'est pas exhaustif, nous donnant même le contre-exemple `(_, _, false, false, false)`. Or, une « simple » analyse des gardes peut prouver que ce cas n'est pas atteignable. Le problème est que cette analyse n'est pas toujours réalisable statiquement. Il suffit en effet d'introduire une expression suffisamment compliquée (*i.e.* appel de fonction, boucle `while` ou encore déclaration `let rec`) pour rendre indécidable notre problème. Il s'agit du théorème de Rice [18]. Ainsi, lorsque l'algorithme `usefulness` nous indique que le filtrage testé n'est pas exhaustif, nous ne pourrons seulement dire « ce filtrage peut ne pas être exhaustif ».

Bien que le problème soit indécidable, nous pouvons trouver dommage que dans des cas si simples aucune tentative ne soit réalisée. Nous pourrions tenter une vérification automatique de la complétude des clauses `when`. En effet, il est possible de produire des obligations de preuve relatives à la complétude des cas, à la manière du comportement induit par les `behaviors` du langage logique ACSL [2]. Certaines de ces obligations de preuves ainsi générées peuvent être prouvées automatiquement par un solveur SMT dans un certain nombre de cas. Le but généré par l'exemple précédent serait `goal g: forall x, y : int. x = y or x < y or x > y` et il est trivialement prouvé par Alt-Ergo [4].

2.10.2 Motifs-ou gardés

Gabriel Scherer *et al.* [19] ont décelé un cas ambigu dans la construction du filtrage. Prenons la simple fonction définie par filtrage :

```
let f = function
    | x, _ | _, x when x = 0 -> true
    | _ -> false
```

Ici, le programmeur s'attend à ce que les résultats de `f` sur `(1,0)` et `(0,1)` soient les mêmes, valant tout les deux `true`. Or ce n'est pas le cas. La subtilité réside dans le fait que l'on retrouve dans ce filtrage un motif-ou sur une ligne contenant un `when` et que les paires `(1,0)` et `(0,1)` peuvent être filtrées par les deux côtés de ce motif-ou. La différence se trouve donc dans l'affectation de la variable `x`. La sémantique du langage OCaml est claire sur ce point : si un motif-ou se trouve sur une ligne alors le premier sous-motif filtrant le paramètre est utilisé pour l'affectation des variables. Dans notre cas les paires pouvant être filtrées par le motif de gauche `(x,_)`, l'affectation correspondante est nécessairement appliquée et si l'évaluation de la clause `when` n'est pas satisfaite alors nous passons directement à la prochaine ligne (sans tester les autres affectations). Ainsi, les valeurs de nos exemples sont `f (1,0) = false` et `f (0,1) = true`. Pour pallier cette ambigüité, nous avons choisi d'interdire dans GOSPEL une telle construction. Une erreur est donc levée lorsqu'un motif-ou se trouve sous une ligne gardée.

3 Revue de la littérature

Construction historique des langages ML, le filtrage par motifs envahit le monde des langages de programmation généralistes depuis plusieurs années. Nous pouvons en effet retrouver cette construction en Rust [9] ou en Scala [15], mais aussi en Python [11] depuis la récente version 3.10. La compilation efficace du filtrage est un problème bien documenté, étant étudié depuis les années 80, dont le précurseur est Lennart Augustsson [1]. Dans ce contexte de langages de programmation, la compilation est le point à développer. Ainsi, Luc Maranget a travaillé la question d'efficacité de la compilation du filtrage [14], mais a aussi proposé un algorithme de détection de non-exhaustivité pour le `match-with` d'OCaml [12, 13]. Cependant, bien que produire un avertissement lorsqu'un filtrage n'est pas exhaustif rend service au programmeur cela n'est pas une nécessité : au pire le programme échouera avec une sortie du type `Match_failure`. En revanche dans le cadre d'un langage logique, nous avons du mal à définir le sens d'un filtrage non exhaustif. L'utilisation d'un tel algorithme est donc indispensable. De plus, nous nous sommes inspirés du travail de John Reppy et Mona Zahir [17], proposant une transformation de la matrice de filtrage pour incorporer la compilation des clauses `when` dans le cas général.

D'autre part, plusieurs langages de spécification formelle sont munis d'une construction plus ou moins proche du filtrage de GOSPEL. Par exemple, le langage ACSL développé par Patrick Baudin *et al.* [2], le langage WhyML développé par Jean-Christophe Filliâtre et Andrei Paskevich [7] et le langage VeriFast développé par Bart Jacobs *et al.* [8]. En effet, le langage ACSL propose une manière de raisonner par cas à l'aide de `behaviors`. De plus, une construction de filtrage polymorphe semble être en cours de développement, mais reste non intégré au noyau Frama-C [5] et mal documentée à l'heure actuelle. Pour le langage WhyML, la notion de filtrage existe dans la logique et la vérification d'exhaustivité est réalisée au typage. Cependant, l'algorithme implémenté est adapté d'un algorithme de compilation du filtrage et n'est pas le même que celui que nous avons choisi ici. Ceci conduit à une vérification d'exhaustivité plus efficace dans GOSPEL que WhyML. Enfin, le langage VeriFast contient une construction `switch` mais celle-ci n'autorise aucun motif profond (pas de motifs-ou ni de motifs imbriqués par exemple). Cette construction est donc significativement moins expressive que le filtrage *à la* ML que nous avons implémenté ici.

4 Conclusion et perspectives

Dans un premier temps, nous avons vu comment le problème de l'exhaustivité du filtrage peut être résolu grâce à l'algorithme `usefulness`. Nous l'avons implémenté et étendu aux clauses `when` dans le code de GOSPEL. Ensuite, nous nous sommes bien assurés de la correction de `usefulness`, aussi bien d'un point de vue théorique avec les preuves de correction et terminaison que d'un point de vue pratique avec cette grande phase de vérification de correspondance avec les résultats du compilateur OCaml utilisé comme oracle sur des tests aléatoires.

Nous pourrions d'une part prolonger ce travail en effectuant une analyse approfondie sur la présence d'éventuelle redondance dans les sous-motifs d'un filtrage en suivant notamment la piste proposée par Luc Maranget [12, 13]. D'autre part, l'usage de solveur SMT pour permettre une plus fine analyse des clauses `when` est aussi une piste intéressante, car elle pourrait permettre parfois d'écrire un filtrage plus élégant.

Je remercie très sincèrement mes encadrants de stage, Jean-Christophe et Clément pour le temps qu'ils m'ont accordé et pour tout l'intérêt qu'ils ont porté à ce travail. Je remercie également les membres de l'équipe Toccata pour leur accueil sympathique et chaleureux.

Références

[1] L. Augustsson. Compiling pattern matching. In *Conference on Functional Programming Languages and Computer Architecture*, pages 368–381. Springer, 1985.

[2] P. Baudin, J.-C. Filliâtre, C. Marché, B. Monate, Y. Moy, and V. Prevosto. Acsl : Ansi c specification language. *CEA-LIST, Saclay, France, Tech. Rep. v1*, 2, 2008.

[3] A. Charguéraud, J.-C. Filliâtre, C. Lourenço, and M. Pereira. Gospel—providing ocaml with a formal specification language. In *International Symposium on Formal Methods*, pages 484–501. Springer, 2019.

[4] S. Conchon, A. Coquereau, M. Iguernlala, and A. Mebsout. Alt-ergo 2.2. In *SMT Workshop : International Workshop on Satisfiability Modulo Theories*, 2018.

[5] P. Cuoq, F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski. Frama-c. In *International conference on software engineering and formal methods*, pages 233–247. Springer, 2012.

[6] J.-C. Filliâtre and C. Pascutto. Ortac : Runtime assertion checking for ocaml (tool paper). In *International Conference on Runtime Verification*, pages 244–253. Springer, 2021.

[7] J.-C. Filliâtre and A. Paskevich. Why3—where programs meet provers. In *European symposium on programming*, pages 125–128. Springer, 2013.

[8] B. Jacobs, J. Smans, P. Philippaerts, F. Vogels, W. Penninckx, and F. Piessens. Verifast : A powerful, sound, predictable, fast verifier for c and java. In *NASA formal methods symposium*, pages 41–55. Springer, 2011.

[9] S. Klabnik and C. Nichols. *The Rust Programming Language (Covers Rust 2018)*. No Starch Press, 2019.

[10] D. E. Knuth. *Art of computer programming, volume 2 : Seminumerical algorithms*, pages 124–125. Addison-Wesley Professional, 2014.

[11] T. Kohn, G. van Rossum, G. B. Bucher II, and I. Levkivskyi. Dynamic pattern matching with python. In *Proceedings of the 16th ACM SIGPLAN International Symposium on Dynamic Languages*, pages 85–98, 2020.

[12] L. Maranget. Les avertissements du filtrage. In *JFLA*, pages 3–20, 2003.

[13] L. Maranget. Warnings for pattern matching. *Journal of Functional Programming*, 17(3) :387–421, 2007.

[14] L. Maranget. Compiling pattern matching to good decision trees. In *Proceedings of the 2008 ACM SIGPLAN workshop on ML*, pages 35–46, 2008.

[15] M. Odersky, P. Altherr, V. Cremet, B. Emir, S. Maneth, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, and M. Zenger. An overview of the scala programming language, 2004.

[16] M. Pereira and A. Ravara. Cameleer : a deductive verification tool for ocaml. In *International Conference on Computer Aided Verification*, pages 677–689. Springer, 2021.

[17] J. Reppy and M. Zahir. Compiling Successor ML pattern guards. In *Proceedings of the 2019 ACM SIGPLAN ML Family Workshop*, Aug. 2019.

[18] H. G. Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, 74(2) :358–366, 1953.

[19] G. Scherer, L. Maranget, and T. Réfis. Ambiguous pattern variables. In *OCaml 2016 : The OCaml Users and Developers Workshop*, page 2, 2016.

[20] R. Sekar, R. Ramesh, and I. Ramakrishnan. Adaptive pattern matching. In *International Colloquium on Automata, Languages, and Programming*, pages 247–260. Springer, 1992.