Les avertissements du filtrage

L. Maranget[1]

1: Inria Rocquencourt
78153 Le Chesnay Cedex, France
`Luc.Maranget@inria.fr`

Résumé

Cet article propose une définition intrinsèque des anomalies de filtrage de ML que sont la clause inutile et le filtrage non-exhaustif, ainsi qu'un algorithme permettant de les signaler au programmeur.

1. Introduction

Le filtrage est certainement l'un des traits distinctifs des langages de la famille ML et les programmeurs y ont largement recours. Le filtrage permet de raisonner (et de programmer) selon le schéma de la définition par cas. Ce style de raisonnement appelle deux contrôles élémentaires : a-t-on bien traité tous les cas possibles, et un cas n'est-il pas couvert par d'autres ? En termes de filtrage, on cherche à écrire des filtrages exhaustifs et dont toutes les clauses sont utiles. En vérifiant ces propriétés, un compilateur ML fournit une aide puissante à la détection d'erreurs de programmation. Cet effort du compilateur est d'autant plus justifié, que lors de la modification d'un programme, on est parfois conduit à introduire ce type d'erreurs très loin de la partie modifiée.

Il existe deux grandes catégories d'algorithmes de compilation du filtrage, que l'on caractérise par leurs cibles : arbres de décision ou automates avec retour en arrière. Les arbres de décision permettent d'effectuer les vérifications lors de la compilation du filtrage. En effet, ces arbres ne contiennent pas de code mort. Mais il y a une bonne raison, comme le fait le compilateur Objective Caml, d'adopter le second style de compilation du filtrage : alors que les arbres de décision peuvent avoir une taille exponentielle en fonction de la taille du filtrage compilé, la taille des automates avec retour en arrière est linéaire. Dès lors, les filtrages doivent être vérifiés par des algorithmes spécifiques, invoqués avant la compilation du filtrage. Cet article décrit ces algorithmes tels que réalisés dans le compilateur Objective Caml.

La section 2 rappelle le contexte théorique du filtrage. Puis la section 3 introduit le problème de la clause utile auquel se ramènent les deux questions de l'exhaustivité et de la clause inutile, et présente un algorithme $\mathcal{U}$ pour le résoudre. L'algorithme $\mathcal{U}$ est ensuite spécialisé, afin de produire un exemple de valeur non-filtrée dans le cas de l'exhaustivité (section 4) et de signaler les alternatives de motifs-ou inutiles (section 5). Nous concluons après un examen partiel de l'efficacité de l'algorithme $\mathcal{U}$.

2. Un peu de formalisme

Les motifs sont des termes définis sur certaines signatures. Les signatures sont introduites par les définition de type de Caml. Par exemple :

```
type t = Nil | One of int | Cons of int * t
```

Les motifs de type `t` seront donc construits à partir d'un ensemble de variables, et des trois constructeurs `Nil` (d'arité zéro ou constructeur *constant*), `One` (unaire) et `Cons` (binaire).

La plupart des types de Caml peuvent être définis de cette façon. Par exemple, les booléens sont un type à deux constructeurs constants, les entiers un type possédant une infinité (ou énormément) de constructeurs constants, les paires sont un type à un constructeur binaire, etc.

En bref, on définit motifs et valeurs ainsi :

$p ::=$	**Motifs**	$v ::=$	**Valeurs**
$\_$	attrape-tout	$c(v_1, v_2, \ldots, v_a)$	
$c(p_1, p_2, \ldots, p_a)$	motif construit		
$(p_1 \mid p_2)$	motif-ou		

Les motifs construits comprennent les constantes (cas $a = 0$). On notera surtout la présence de l'attrape-tout « $\_$ » qui est une variable dont le nom n'a pas d'importance et qui, dans cet article, remplace toutes les variables. On notera aussi que les valeurs sont simplement des motifs sans variable ni motif-ou.

Les motifs filtrent les valeurs au sens qu'ils décrivent le haut d'une valeur. On peut définir cette relation, notée $\preceq$, par l'existence d'une substitution ou inductivement (les deux définitions étant équivalentes dans le cas des motifs linéaires).

Définition 1 (Relation de filtrage)

$$
\begin{array}{rcll}
\_ & \preceq & v & \\
(p_1 \mid p_2) & \preceq & v & \textit{ssi } p_1 \preceq v \textit{ ou } p_2 \preceq v \\
c(p_1, \ldots, p_a) & \preceq & c(v_1, \ldots, v_a) & \textit{ssi } (p_1 \cdots p_a) \preceq (v_1 \cdots v_a) \\
(p_1 \cdots p_a) & \preceq & (v_1 \cdots v_a) & \textit{ssi } p_i \preceq v_i, \forall i \in [1..a]
\end{array}
$$

Lorsque le motif p filtre la valeur v ont dit aussi que v est une instance *de p.*

Si on convient qu'un motif représente l'ensemble de ses instances, le motif-ou exprime simplement la réunion de ces ensembles. Il faut noter une fois pour toutes que le filtrage tient compte du typage. De notre point de vue, ce sont les motifs qui imposent l'ensemble des valeurs où chercher leurs instances. Par exemple, la question de savoir si une valeur $(v_1,\ v_2)$ est ou n'est pas une instance du motif `(1,_)` ne se pose que si v_1 est un entier.

La notation utilisée dans le dernier cas de la définition du filtrage définit le filtrage des vecteurs de valeurs par les vecteurs de motifs. Cette notation pratique sera largement utilisée dans cet article. Et je commence tout de suite en définissant le filtrage de ML dans ce formalisme.

Définition 2 (Filtrage de ML) *Soient P, matrice de motifs et $\vec{v} = (v_1 \cdots v_n)$, vecteur de valeurs, où n est également la largeur de P. La i-ème ligne de P filtre $\vec{v}$, si et seulement si les deux conditions suivantes sont satisfaites.*

1. $(p^i_1 \cdots p^i_n) \preceq (v_1 \cdots v_n)$
2. $\forall j < i,\ (p^j_1 \cdots p^j_n) \not\preceq (v_1 \cdots v_n)$

Autrement dit, le vecteur $\vec{v}$ filtre la première ligne possible en partant du haut de la matrice P.

Si on ne tient pas compte de l'ordre des lignes on peut définir plus globalement le filtrage par une matrice de motifs.

Définition 3 (Filtrage par une matrice) *Soient P matrice de motifs de taille $n \times m$ et $\vec{v} = (v_1 \cdots v_n)$ vecteur de valeurs. La matrice P filtre le vecteur $\vec{v}$, noté abusivement $P \preceq \vec{v}$, si et seulement si il existe un numéro de ligne i, avec :*

$$(p^i_1 \cdots p^i_n) \preceq (v_1 \cdots v_n)$$

Dans ce cadre une matrice représente simplement la réunion des instances de ses lignes.

En fait, si on s'intéresse uniquement à l'existence d'au moins une ligne de la matrice P filtrant le vecteur $\vec{v}$ la règle de priorité (règle 2. de la définition 2) n'est plus utile.

Lemme 1 *Soient P matrice de motifs de taille $n \times m$ et $\vec{v}$ vecteur de valeurs de taille n. Alors il existe une ligne de P qui filtre $\vec{v}$ au sens de la définition 2, si et seulement si P filtre v au sens de la définition 3*

Démonstration : Le sens direct est évident. Pour démontrer le sens réciproque, il suffit de choisir le plus petit numéro de ligne i tel que $\vec{p^i} \preceq \vec{v}$.

Enfin la relation de filtrage proprement dite $\preceq$ induit une relation entre les motifs eux-mêmes.

Définition 4 (Précision des motifs) *Le motif p est* moins précis *que le motif q, noté $p \preceq q$, quand toutes les instances de q sont des instances de p.*

La relation de précision est une relation d'ordre partiel (et bien fondée) sur l'ensemble des motifs, en fait c'est l'anti-inclusion « $\supseteq$ ». À cause du typage, la précision n'a pas de définition inductive simple. En effet on a par exemple `(_,_)` $\preceq$ `_`, mais pas `One _` $\preceq$ `_` (puisque la valeur `Nil` est une instance du motif `_` sans être une instance du motif `One _`). Les choses se compliquent encore un peu avec les motifs-ou. Toutefois, comme on ne tardera pas à la voir, la relation de précision est aisément décidable.

3. La clause utile

Les deux questions de l'exhaustivité et de la clause inutile peuvent s'exprimer dans le cadre du filtrage par des matrices.

Définition 5 (Exhaustivité) *Soit P, matrice de motifs. La matrice P est exhaustive si, pour tout vecteur $\vec{v}$ du type approprié, il existe une ligne de P filtrant $\vec{v}$ au sens de la définition 2.*

Définition 6 (Clause inutile) *Soit P, matrice de motifs. La i-ème ligne de P est inutile si il n'existe pas d'instances de cette ligne au sens de la définition 2.*

On remarquera ici que la première ligne d'une matrice est toujours utile.

Ces deux questions sont en fait deux instances du même problème.

Définition 7 (Clause utile) *Soient P, matrice de motifs de taille $n \times m$ et $\vec{q}$, vecteur de motifs de taille n. Le vecteur $\vec{q}$ est utile relativement à la matrice P, si et seulement si il existe une instance de $\vec{q}$ qui n'est pas une instance de P. Soit, la proposition $\mathcal{U}(P, \vec{q})$ suivante est vraie.*

$$\exists \vec{v}, P \npreceq \vec{v} \land \vec{q} \preceq \vec{v}.$$

Proposition 1 *Les problèmes de l'exhaustivité et de la clause inutile s'expriment à l'aide du problème de la clause utile.*

1. *La matrice P est exhaustive, si et seulement si $\mathcal{U}(P, (\_\ \_ \cdots \_))$ est faux.*
2. *La i-ème ligne de la la matrice P est inutile si et seulement si $\mathcal{U}(Q, \vec{p}^{\,i})$ est faux. Où la matrice Q regroupe les lignes précédant la i-ème dans P.*

La proposition est en fait facile, on remarquera quand même que l'on passe de la définition du filtrage de ML à celle du filtrage par une matrice, ce que justifie le lemme 1.

3.1. Résolution du problème de la clause utile

L'algorithme donné dans cette section sous forme d'une fonction $\mathcal{U}$ définie récursivement est similaire à la compilation d'un filtrage de ML vers les arbres de décision ([5] donne une bonne présentation de cet algorithme assez ancien). Soient donc la matrice P et le vecteur $\vec{q}$.

1. Considérons d'abord le cas de base. Il n'y a pas de colonnes ($n = 0$), alors $\vec{q}$ est le vecteur de taille nulle, noté () et la réponse dépend du nombre de lignes m de P.

   Si m est nul alors, alors P est la matrice vide, sans lignes et notée $\emptyset$. Dès lors, $\vec{q}$ filtre bien le vecteur de taille nulle, qui ne peut être filtré par une ligne de P puisque P ne possède aucune ligne. Sinon, m n'est pas nul, c'est-à-dire que P possède au moins une ligne qui filtre (), la seule instance possible de $\vec{q}$. Soit encore :

   $$\mathcal{U}(\emptyset, ()) = \mathit{vrai} \qquad\qquad \mathcal{U}(\begin{pmatrix} \\ \vdots \\ \\ \end{pmatrix}, ()) = \mathit{faux}$$

   On peut noter au passage que si P n'a pas de lignes, alors le nombre de colonnes de $\vec{q}$ importe peu. On a toujours $\mathcal{U}(\emptyset, \vec{q}) = \mathit{vrai}$.

2. Sinon ($n > 0$), il y a des colonnes. On en choisit une, par exemple, pour simplifier, la première. Il y a alors trois sous-cas selon la nature du motif q_1.

   (a) Le motif q_1 est construit, c'est-à-dire $q_1 = c(r_1, \ldots, r_a)$. Alors, toute instance $\vec{v}$ de $\vec{q}$ possède nécessairement une première composante de la forme $v_1 = c(w_1, \ldots, w_a)$. On peut extraire de la matrice P une nouvelle matrice $\mathcal{S}(c, P)$ telle que les prédicats $P \preceq (c(w_1, \ldots, w_a) \cdots v_n)$ et $\mathcal{S}(c, P) \preceq (w_1 \cdots w_a\ v_2 \cdots v_n)$ soient équivalents. La matrice $\mathcal{S}(c, P)$ est de largeur $a + n - 1$ et ses lignes sont définies à partir de celles de P selon leur première composante p_1^i.

p_1^i	$\mathcal{S}(c, P)$
$c(t_1, \ldots, t_a)$	$t_1 \cdots t_a \quad p_2^i \cdots \quad p_n^i$
$c'(t_1, \ldots, t_{a'})\ (c' \neq c)$	pas de ligne
$\_$	$\_ \cdots \_ \quad p_2^i \cdots \quad p_n^i$
$(t_1 \mid t_2)$	$\mathcal{S}(c, \begin{pmatrix} t_1 & p_2^i \cdots p_n^i \\ t_2 & p_2^i \cdots p_n^i \end{pmatrix})$

   On peut alors simplement poser :

   $$\mathcal{U}(P, (c(r_1, \ldots, r_a)\ q_2 \cdots q_n)) = \mathcal{U}(\mathcal{S}(c, P), (r_1 \cdots r_a\ q_2 \cdots q_n))$$

   (b) Le motif q_1 est un attrape-tout. Soient $\Sigma = \{c_1, c_2, \ldots, c_z\}$ les constructeurs qui apparaissent en tête des premiers motifs des lignes de P (et de leurs arguments si ce sont des motifs-ou). On note a_k l'arité du constructeur c_k. En raison du typage, on sait que toute instance de $\vec{q}$ possède une première composante v_1, qui est telle que :
   - v_1 est de la forme $c_k(w_1, \ldots, w_{a_k})$,
   - ou bien, v_1 est de la forme $c(w_1, \ldots, w_a)$, où c est un autre constructeur dans la signature des constructeurs de Σ si il en existe.

Pour traiter ce dernier cas on construira une « *matrice par défaut* », $\mathcal{D}(P)$, de largeur $n-1$:

p_1^i	$\mathcal{D}(P)$
$c_k(t_1, \ldots, t_{a_k})$	pas de ligne
$\_$	$p_2^i \cdots p_n^i$
$(t_1 \mid t_2)$	$\mathcal{D}(\begin{pmatrix} t_1 & p_2^i \cdots p_n^i \\ t_2 & p_2^i \cdots p_n^i \end{pmatrix})$

Ce calcul s'applique encore quand Σ est vide, c'est-à-dire quand la première colonne de la matrice P est constituée seulement d'attrape-tout. Le calcul de $\mathcal{D}(P)$ revient alors a supprimer cette première colonne.

On a ensuite :

$$\mathcal{U}(P, (\_\ q_2 \cdots q_n)) = \left(\bigvee_{k=1}^{z} \mathcal{U}(\mathcal{S}(c_k, P), (\underbrace{\_ \cdots \_}_{a_k \text{ fois}}\ q_2 \cdots q_n)) \right) \vee \mathcal{U}(\mathcal{D}(P), (q_2 \cdots q_n))$$

Le dernier terme de cette disjonction est absent lorsque Σ forme une signature complète de constructeurs. Tandis que c'est le premier terme qui disparaît lorsque Σ est vide.

(c) Le motif q_1 est un motif-ou $(r_1 \mid r_2)$. On a alors :

$$\mathcal{U}(P, ((r_1 \mid r_2)\ q_2 \cdots q_n)) = \mathcal{U}(P, (r_1\ q_2 \cdots q_n)) \vee \mathcal{U}(P, (r_2\ q_2 \cdots q_n))$$

L'algorithme $\mathcal{U}$ permet bien évidemment de résoudre les problèmes de l'exhaustivité et de la clause inutile. Pour fixer définitivement les idées, étant donné un filtrage :

`match ... with` p_1 `->` e_1 `|` p_2 `->` e_2 `| ... |` p_m `->` e_m

On vérifie l'exhaustivité par l'appel :

$$\mathcal{U}(\begin{pmatrix} p_1 \\ p_2 \\ \vdots \\ p_m \end{pmatrix}, (\_))$$

Tandis que pour vérifier l'utilité de la i-ème clause, on calcule :

$$\mathcal{U}(\begin{pmatrix} p_1 \\ p_2 \\ \vdots \\ p_{i-1} \end{pmatrix}, (p_i))$$

Mais $\mathcal{U}$ peut aussi être utilisé pour décider la relation de précision entre deux motifs, on a $p \preceq q$ si et seulement si $\mathcal{U}((p), (q))$ est faux. On pourrait opérer plus directement, mais une fois écrit le code de $\mathcal{U}$, cette équivalence est bien pratique.

4. Spécialisation de $\mathcal{U}$ pour l'exhaustivité

Avec le temps, les utilisateurs d'Objective Caml, tout en se montrant satisfaits d'avertissements exacts ont demandé plus. En cas de filtrage non-exhaustif, ils réclamaient un « *exemple* ». Un tel exemple est *a priori* une instance, c'est-à-dire une valeur. Toutefois, il est plus commode de donner un motif dont toutes les instances sont des exemples, c'est-à-dire un motif dont aucune instance n'est filtrée. On peut alors compléter le filtrage par ce motif, dans l'espoir que le filtrage complété est exhaustif. Pour prendre un exemple facile considérons le programme :

```
let nilp = function [] -> true
Warning: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
_::_
```

Le filtrage n'est pas exhaustif et les instance du motif `_::_` sont toutes non-filtrées. Ici, en ajoutant un cas, on obtient un filtrage exhaustif.

Les exemples de valeurs non-filtrées se calculent par une extension simple de l'algorithme $\mathcal{U}$. L'extension est simple car $\mathcal{U}$ vérifie qu'un certain ensemble n'est pas vide en calculant implicitement un de ses éléments. Le nouvel algorithme $\mathcal{I}$ prend en argument une matrice P et un entier n, où n est le nombre de colonnes de P. En effet, dans le cas de la vérification d'exhaustivité, l'argument $\vec{q}$ de $\mathcal{U}$ est toujours un vecteur d'attrape-tout, on peut donc le remplacer par sa taille n. L'algorithme $\mathcal{I}$ renvoie un vecteur de motifs $\vec{p}$ de taille n, tel qu'aucune instance de $\vec{p}$ n'est filtrée par P. Si un tel vecteur de motifs n'existe pas, $\mathcal{I}(P, n)$ vaut $\perp$. $\mathcal{I}$ se calcule comme $\mathcal{U}$ par cas selon le nombre de colonnes n.

1. Si n est nul, on retrouve les deux cas de base.

$$\mathcal{I}(\emptyset, 0) = () \qquad\qquad \mathcal{I}(\begin{pmatrix} \\ \vdots \\ \\ \end{pmatrix}, 0) = \perp$$

   Plus généralement, $\mathcal{I}(\emptyset, n)$ est le vecteur composé de n attrape-tout.

2. Si n n'est pas nul, on calcule les matrices $\mathcal{S}(c_k, P)$ et éventuellement $\mathcal{D}(P)$, comme à la section 3.1. Les appels récursifs possibles sont donc $\mathcal{I}(\mathcal{S}(c_k, P), a_k + n - 1)$, pour c_k pris dans Σ et éventuellement $\mathcal{I}(\mathcal{D}(P), n-1)$ Si tous ces appels renvoient $\perp$, alors $\mathcal{I}(P, n)$ est également $\perp$. Sinon, il y a plusieurs possibilités, dont les deux premières ne s'excluent pas nécessairement (ce qui entraîne une indétermination de l'exemple de valeur non-filtrée).
   - Si l'un des appels récursifs sur $\mathcal{S}(c_k, P)$ renvoie le vecteur $(r_1 \cdots r_{a_k}\ p_2 \cdots p_n)$ on pourra poser $\mathcal{I}(P, n) = (c_k(r_1, \ldots, r_{a_k})\ p_2 \cdots p_n)$. Le choix du c_k parmi ceux qui conviennent étant bien entendu libre.
   - On peut aussi, si l'appel récursif sur la matrice par défaut renvoie le vecteur $(p_2 \cdots p_n)$ poser $\mathcal{I}(P, n) = (c(\_, \ldots, \_)\ p_2 \cdots p_n)$, où c est un constructeur de la signature des c_k qui n'est pas un c_k. Dans le cas d'une signature finie, on peut même utiliser un motif-ou regroupant tous les constructeurs complétant la signature des c_k.
   - Enfin, si Σ est vide, c'est-à-dire si la première ligne de P est entièrement constituée d'attrape-tout, alors cette première ligne n'a pas d'importance et on a : $\mathcal{I}(P, n) = (\_\ p_2 \cdots p_n)$.

   Dans ces trois cas, on vérifie facilement en revenant aux définitions que $\mathcal{I}$ renvoie bien un vecteur de motifs dont aucune instance n'est filtrée par P.

L'algorithme $\mathcal{I}$ n'est pas déterministe, ainsi vérifions l'exhaustivité du filtrage

```
match ... with One 1 -> ...
```

On calcule donc $\mathcal{I}((\texttt{One 1}), 1)$. Les motifs `One 0` et `(Nil|Cons (_,_))` sont des réponses possibles. En effet posons $P = (\texttt{One 1})$. On peut d'une part considérer le filtrage de P par le constructeur `One`, soit $\mathcal{S}(\texttt{One}, P) = (\texttt{1})$, matrice qui sera à son tour décomposée en $\mathcal{S}(\texttt{1}, (\texttt{1})) = ()$ et $\mathcal{D}((\texttt{1})) = \emptyset$. Soit, en remontant les appels, $\mathcal{I}(\emptyset, 0) = ()$ et puis par exemple $\mathcal{I}((\texttt{1}), 1) = \texttt{0}$, et donc finalement $\mathcal{I}(P, 1) = (\texttt{One 0})$. D'autre part, en sélectionnant la matrice par défaut de $(\texttt{One 1})$ ici vide, il vient plus directement $\mathcal{I}(P, 1) = (\texttt{(Nil|Cons (\_,\_))})$.

Il s'agit d'ailleurs du résultat le plus probablement attendu, ce qui donne une première raison d'essayer de satisfaire les matrices par défaut avant les autres (ce choix peut aussi se justifier par l'efficacité, cf. section 6.2). On peut s'attendre à ce que les utilisateurs les plus exigeants demandent une information plus complète (ici ce sera (`One 0|(Nil|Cons (_,_))`)). C'est je crois déraisonnable, les motifs produits pouvant être gros au point de devenir difficiles à interpréter.

5. Spécialisation de $\mathcal{U}$ pour la clause inutile

En l'absence de motifs-ou, l'algorithme $\mathcal{U}$, qui signale simplement les clauses inutiles, suffit. En effet, on voit mal quelle autre information concise et précise donner au programmeur qui a écrit une clause inutile et se contente en général de la supprimer avant de recompiler.

5.1. Insuffisance de la clause inutile

Mais les motifs-ou conduisent à un autre style d'erreur que $\mathcal{U}$ ne signale pas. Un programmeur cherchant à détecter les listes de de type `t` dont le premier élément est `1` pourrait écrire :

```
let f = function
| One 1 | Cons (1,_) -> true
| _ -> false
```

À la réflexion notre programmeur se dit qu'il n'est pas protégé contre l'ajout d'un nouveau constructeur au type `t`. Il réécrit :

```
let f = function
| One 1 | Cons (1,_) -> true
| Nil | One _ | Cons (_,_) -> false
```

Plus tard un ami lui signale que les motifs-ou peuvent contenir des variables. Ravi, il réécrit encore :

```
let f = function
| One x | Cons (x,_) -> x=1
| Nil | One _ | Cons (_,_) -> false
```

Il semble bien que le dernier motif soit un peu compliqué, Et en effet, le code suivant est plus concis et rigoureusement équivalent.

```
let f = function
| One x | Cons (x,_) -> x=1
| Nil  -> false
```

Malheureusement, le compilateur reste muet face au « *mauvais* » code. Alors qu'il aurait dénoncé deux clauses inutiles à la fin de la fonction `f` suivante.

```
let f = function
| One x | Cons (x,_) -> x=1
| Nil -> false | One _ -> false | Cons (_,_) -> false
```

L'exemple montre bien où le bât blesse. La *clause* `Nil | One _ | Cons (_,_) -> false` est utile (à cause de `Nil`) mais elle contient des *motifs* inutiles, puisque l'on peut supprimer ceux-ci sans rien changer au filtrage. Par ailleurs, une définition plus positive des motifs inutiles se fait facilement par un recours à l'expansion des motifs-ou.

En raison de la possibilité de motifs-ou nombreux, on a intérêt à définir l'expansion d'un seul motif-ou, afin d'éviter la production d'un nombre exponentiel de motifs. Un exemple suffit pour illustrer ce propos.

```
let f = function
| (1|2), (3|4), (6|7), ..., (2k−1|2k) -> true
| _ -> false
```

Si l'on souhaite s'assurer de d'utilité des deux arguments du motif-ou $(2k-1|2k)$, il suffit de n'expanser que ce dernier.

```
let f = function
| (1|2), (3|4), (6|7), ..., 2k−1 -> true
| (1|2), (3|4), (6|7), ..., 2k -> true
| _ -> false
```

L'expansion se fait en considérant que l'alternative de gauche est plus prioritaire que l'alternative de droite. Cela permet de donner un sens clair aux motifs inutiles dans les cas tangents suivants :

```
let f1 = function (1|_) -> true
and f2 = function (_|1) -> true
and f3 = function (1|1) -> true
and f4 = function (_|_) -> true
```

L'expansion montre dans quel sens le second argument du motif-ou est utile dans le premier exemple et inutile dans les trois derniers :

```
let f1 = function 1 -> true | _ -> true
and f2 = function _ -> true | 1 -> true
and f3 = function 1 -> true | 1 -> true
and f4 = function _ -> true | _ -> true
```

Ainsi, même si l'ordre de vérification des alternatives d'un motif-ou est indéterminé (voir la définition 1), le jugement porté par le compilateur sur l'utilité de ces alternatives demande de lever cette indétermination. Lever l'indétermination dans la définition du filtrage par les motifs me semble gratuit, car cet ordre n'a pas d'influence sur la définition 2 du filtrage par les lignes d'une matrice. Cependant, dans un souci de cohérence, le compilateur du filtrage [3] devrait tenir compte de cet ordre gauche-droite des alternatives des motifs-ou.

5.2. Règles pour un calcul des motifs inutiles

Il paraît plus facile de présenter la recherche des sous-motifs inutiles d'abord dans le cas d'un seul motif-ou. Soit donc un filtrage $P, \vec{q}$ tel que q_1 est le motif-ou $(r_1 \,|\, r_2)$. On se propose de décider entre les quatre possibilités, r_1 et r_2 sont tous deux utiles, r_1 est seul inutile, r_2 est seul inutile, ou finalement r_1 et r_2 sont tous les deux inutiles et donc $\vec{q}$ est inutile relativement à P. On cherche donc à concevoir une fonction $\mathcal{U}'(P, \vec{q})$ qui renvoie un ensemble de motifs inutiles, à savoir $\emptyset$ dans le premier cas, $\{r_1\}$ dans le deuxième, $\{r_2\}$ dans le troisième, et un ensemble distingué $\top$ dans le quatrième et dernier cas.

On définit alors deux filtrage expansés $P', \vec{q}\,'$ et $P'', \vec{q}\,''$ avec

$$P' = P, \quad \vec{q}\,' = (r_1\ q_2 \cdots q_n) \qquad \text{et} \qquad P'' = P@\vec{q}\,', \quad \vec{q}\,'' = (r_2\ q_2 \cdots q_n)$$

Où $P@\vec{q}\,'$ signifie simplement l'ajout de la ligne $\vec{q}\,'$ à celles de la matrice P. On calcule l'utilité des deux expansions en utilisant $\mathcal{U}$. Si aucune des expansions ne contient de motif-ou, les résultats produits sont exacts. On les note E' et E'', en choisissant logiquement de coder *vrai* par $\emptyset$ et *faux* par $\top$, on doit ensuite combiner E' et E'' selon une règle qui s'apparente à une union ensembliste et que l'on note $E' \cup E''$:

E'	E''	$E' \cup E''$
$\emptyset$	$\emptyset$	$\emptyset$
$\top$	$\top$	$\top$
$\emptyset$	$\top$	$\{r_2\}$
$\top$	$\emptyset$	$\{r_1\}$

Ce tableau renvoie directement à la définition du motif inutile par expansion en clause inutile.

Supposons maintenant que r_1 et r_2 sont des motifs-ou. Alors, on peut en fait appeler récursivement $\mathcal{U}'$ sur les expansions et combiner les résultats additionnels de la façon suivante :

E'	E''	$E' \cup E''$
$\emptyset$	$\{r'', \ldots\}$	$\{r'', \ldots\}$
$\{r', \ldots\}$	$\emptyset$	$\{r', \ldots\}$
$\top$	$\{r'', \ldots\}$	$\{r_1, r'' \ldots\}$
$\{r', \ldots\}$	$\top$	$\{r_2, r' \ldots\}$
$\{r', \ldots\}$	$\{r'', \ldots\}$	$\{r', \ldots, r'' \ldots\}$

Il s'agit donc bien d'une réunion ensembliste. Le tableau s'explique encore selon la définition des motifs inutiles par expansion. Supposons par exemple $q_1 = ((r_1 \mid r_2) \mid (r_3 \mid r_4))$ On voit alors que les mêmes expansions servent à définir et calculer l'utilité de r_1, r_2, r_3 et r_4. Par exemple, dans le cas de r_4 cette expansion est constituée de la matrice $P@((r_1 \mid r_2)\ q_2 \cdots q_n)@(r_3\ q_2 \cdots q_n)$ et du vecteur $(r_4\ q_2 \cdots q_n)$.

Notons que les motifs éléments de E' et E'' s'entendent comme provenant du source. Deux motifs structurellement égaux mais provenant d'endroits distincts du programme source doivent être considérés comme distincts. Les motifs présents dans E' et E'' sont donc toujours deux à deux distincts puisqu'ils proviennent de parties disjointes du source.

Si $\vec{q}$ contient directement plusieurs motifs-ou, par exemple si q_1 et q_2 sont des motifs-ou, alors on peut procéder comme ci-dessus pour traiter le cas du motif-ou q_1, obtenant ainsi un résultat E_1. Puis, on peut récidiver après échange des premières et deuxièmes colonnes de P et $\vec{q}$, obtenant ainsi un second résultat E_2. Dans les deux cas, on supposera que les éventuelles expansions de motifs-ou pratiquées récursivement ne s'appliquent qu'à la première colonne. Le résultat final est encore une fois la réunion $E_1 \cup E_2$, qui exprime cette fois la combinaison de motifs inutiles issus de deux motifs-ou disjoints. Il faut remarquer que le cas $E_1 = \top$ et $E_2 \neq \top$ (ou le contraire) est ici impossible. En effet, on a bien $\mathcal{U}(P, \vec{q}) = \mathit{faux}$ si et seulement si $\mathcal{U}'(P, \vec{q}) = \top$, et ce indépendamment de la colonne expansée. Plus simplement peut-être, il est clair que si $\vec{q}$ n'est pas utile relativement à P, alors tous les arguments des motifs-ou de $\vec{q}$ sont pareillement inutiles.

Enfin, si P ne contient pas directement de motif-ou, mais si par exemple q_1 est de la forme $c(r_1, \ldots, r_a)$, où par exemple r_1 est un motif-ou. Alors, il suffit de filtrer P et $\vec{q}$ selon le constructeur c, c'est-à-dire de poser :

$$\mathcal{U}'(P, (c(r_1, \ldots, r_a)\ q_2 \cdots q_n)) = \mathcal{U}'(\mathcal{S}(c, P), (r_1 \cdots r_a\ q_2 \cdots q_n))$$

5.3. Calcul des motifs inutiles

Nous sommes maintenant en mesure de présenter l'algorithme complet, tel qu'il est réalisé dans le compilateur Objective Caml. Afin de limiter les expansions dans le cas de plusieurs motifs-ou, cet algorithme opère sur les matrices et vecteurs dont les colonnes sont réparties en trois sous-ensembles. On note cette séparation par « • ». Soit $\mathcal{U}'$ s'applique à de telles matrices et vecteurs pointés notés $P \bullet Q \bullet R$ et $\vec{p} \bullet \vec{q} \bullet \vec{r}$. On retrouve la matrice et le vecteur définissant le filtrage en groupant les sous-ensembles de colonnes, ce que l'un note $P \,\&\, Q \,\&\, R$ et $\vec{p} \,\&\, \vec{q} \,\&\, \vec{r}$. Informellement, P regroupe les colonnes non encore examinées, Q les colonnes à discriminer selon un attrape-tout (les composantes de $\vec{q}$ sont des attrape-tout), et R les colonnes à discriminer selon un motif-ou.

La figure 1 définit quelques opérations utiles sur ces matrices pointées, en supposant qu'il y n colonnes dans P. L'opération de filtrage par un constructeur s'exerce sur le premier sous-ensemble de

FIG. 1 – Opérations sur les matrices pointées

(a) Filtrage par le constructeur c

ligne de $P \bullet Q \bullet R$	ligne(s) de $\mathcal{S}(c, P \bullet Q \bullet R)$
$c(t_1, \ldots, t_a)\ \cdots\ p_n^i \bullet q_1^i \cdots q_k^i \bullet r_1^i \cdots r_z^i$	$t_1 \cdots t_a\ p_2^i \cdots p_n^i \bullet q_1^i \cdots q_k^i \bullet r_1^i \cdots r_z^i$
$\_\ \cdots\ p_n^i \bullet q_1^i \cdots q_k^i \bullet r_1^i \cdots r_z^i$	$\_ \cdots \_\ p_2^i \cdots p_n^i \bullet q_1^i \cdots q_k^i \bullet r_1^i \cdots r_z^i$
$c'(t_1, \ldots, t_{a'})\ \cdots\ p_n^i \bullet q_1^i \cdots q_k^i \bullet r_1^i \cdots r_z^i$	pas de ligne
$(t_1 \mid t_2)\ \cdots\ p_n^i \bullet q_1^i \cdots q_k^i \bullet r_1^i \cdots r_z^i$	$\mathcal{S}\left(c, \begin{pmatrix} t_1 \cdots p_n^i \bullet q_1^i \cdots q_k^i \bullet r_1^i \cdots r_z^i \\ t_2 \cdots p_n^i \bullet q_1^i \cdots q_k^i \bullet r_1^i \cdots r_z^i \end{pmatrix}\right)$

(b) Décalages

ligne de $P \bullet Q \bullet R$
$p_1^i \cdots p_n^i \bullet q_1^i \cdots q_k^i \bullet r_1^i \cdots r_z^i$

ligne de $\Rightarrow_1(P \bullet Q \bullet R)$
$p_2^i \cdots p_n^i \bullet p_1^i\ \ q_1^i \cdots q_k^i \bullet r_1^i \cdots r_z^i$

ligne de $\Rightarrow_2(P \bullet Q \bullet R)$
$p_2^i \cdots p_n^i \bullet q_1^i \cdots q_k^i \bullet p_1^i\ \ r_1^i \cdots r_z^i$

colonnes P, mais elle concerne toutes les lignes de $P \bullet Q \bullet R$. Les deux décalages consistent simplement à transférer la première colonne de P vers Q ou R.

L'algorithme $\mathcal{U}'$ prend en arguments une matrice pointée et un vecteur pointé, et il renvoie un ensemble de motifs inutiles. Pour connaître l'utilité de la i-ème clause d'un filtrage

`match ... with` p_1 `->` e_1 `|` p_2 `->` e_2 `| ... |` p_m `->` e_m

on procède à l'appel initial

$$\mathcal{U}'(\begin{pmatrix} p_1 \\ p_2 \\ \vdots \\ p_{i-1} \end{pmatrix} \bullet \bullet, (p_i) \bullet \bullet)$$

On peut voir $\mathcal{U}'$ comme se décomposant en deux phases, d'abord une phase de recherche des motifs-ou dans p_i, puis une phase de vérification des expansions. Dans l'appel typique $\mathcal{U}'(P \bullet Q \bullet R, \vec{p} \bullet \vec{q} \bullet \vec{r})$, on distingue quatre cas, les trois premiers réalisant la première phase et s'appliquant lorsque P contient des colonnes ($n > 0$).

1. Si p_1 est un motif construit $p_1 = c(t_1, \ldots, t_a)$, alors on pose :

$$\mathcal{U}'(P \bullet Q \bullet R, (c(t_1, \ldots, t_a)\ t_2 \cdots t_n) \bullet \vec{q} \bullet \vec{r}) = \mathcal{U}'(\mathcal{S}(c, P \bullet Q \bullet R), (t_1 \cdots t_a\ p_2 \cdots p_n) \bullet \vec{q} \bullet \vec{r})$$

2. Si p_1 est un attrape-tout, on transfère la première colonne de P dans Q, ce que l'on peut noter :

$$\mathcal{U}'(P \bullet Q \bullet R, (\_\ p_2 \cdots p_n) \bullet \vec{q} \bullet \vec{r}) = \mathcal{U}'(\Rightarrow_1(P \bullet Q \bullet R), \Rightarrow_1(\vec{p} \bullet \vec{q} \bullet \vec{q}))$$

3. Si p_1 est un motif-ou, on transfère la première colonne de P dans R. Soit :

$$\mathcal{U}'(P \bullet Q \bullet R, ((t_1 \mid t_2)\ p_2 \cdots p_n) \bullet \vec{q} \bullet \vec{r}) = \mathcal{U}'(\Rightarrow_2(P \bullet Q \bullet R), \Rightarrow_2(\vec{p} \bullet \vec{q} \bullet \vec{q}))$$

4. Si n est nul, alors la première phase est terminée, $\vec{q}$ regroupe des attrape-tout et $\vec{r}$ regroupe des motifs-ou. On a alors deux sous-cas selon que $\vec{r}$ est de taille nulle ou non.

(a) Donc, si il n'y a en définitive pas de motif-ou, on se contente d'appeler $\mathcal{U}$.

$$\begin{array}{lcll} \mathcal{U}'(\ \bullet Q \bullet\ ,\ \bullet(\_\ \_ \cdots \_) \bullet\ ) & = & \emptyset & \text{si } \mathcal{U}(Q, (\_\ \_ \cdots \_)) = \textit{vrai} \\ \mathcal{U}'(\ \bullet Q \bullet\ ,\ \bullet(\_\ \_ \cdots \_) \bullet\ ) & = & \top & \text{si } \mathcal{U}(Q, (\_\ \_ \cdots \_)) = \textit{faux} \end{array}$$

(b) Sinon, R et $\vec{r}$ ont z colonnes ($z > 0$). Toutes les composantes de $\vec{r}$ sont des motifs-ou à expanser. Soit j indice de colonne dans R et $\vec{r}$, tel que $r_j = (t_1 \mid t_2)$. On note $\vec{r} \setminus j$ le vecteur obtenu en supprimant t_j, de même on note $R \setminus j$ la matrice obtenue en supprimant la j-ème colonne de R. Enfin, on note $[R]_j$ la matrice réduite à la j-ème colonne de R. L'utilité des arguments du motif-ou r_j est donnée par :

$$E_j = \left\{ \begin{array}{l} \mathcal{U}'([R]_j \bullet (R \setminus j) \,\&\, Q \bullet\ , (t_1) \bullet (\vec{r} \setminus j) \,\&\, \vec{q} \bullet\ ) \\ \cup \\ \mathcal{U}'([R]_j @ (t_1) \bullet ((R \setminus j) \,\&\, Q) @ ((\vec{r} \setminus j) \,\&\, \vec{q}) \bullet\ , (t_2) \bullet (\vec{r} \setminus j) \,\&\, \vec{q} \bullet\ ) \end{array} \right.$$

Cette expression compliquée signifiant simplement que l'on recommence la première phase de l'algorithme sur l'expansion du motif-ou r_j, les colonnes correspondant aux autres motifs-ou échappant à toute nouvelle expansion. Les notations obscurcissant peut être un peu l'expansion, on pourra se reporter au cas simple du début de la section 5.2.

Enfin, on combine les résultats des expansions de tous les motifs-ou directement présents dans $\vec{t}$.

$$\mathcal{U}'(\ \bullet Q \bullet R,\ \bullet \vec{q} \bullet \vec{r}) = \bigcup_{j=1}^{z} E_j$$

On peut améliorer légèrement l'algorithme en deux points.

- Lors de l'étape 2, si tous les motifs de la première colonne de P sont des attrape-tout, on efface cette première colonne au lieu de la transférer dans Q. Cela évitera l'effacement répété de cette colonne par tous les appels ultérieurs de $\mathcal{U}$.
- Lors du calcul de la réunion des E_j, on peut renoncer et renvoyer $\top$ dès que le premier E_j calculé vaut $\top$.

6. Performance

6.1. Coût

Pour un filtrage donné comportant m motifs, $\mathcal{I}$ est appelé une fois et $\mathcal{U}'$ est appelé m fois, sur des matrices colonnes de hauteur allant de zéro à $m - 1$. L'efficacité des deux algorithmes $\mathcal{I}$ et $\mathcal{U}'$ repose au final sur celle de $\mathcal{U}$. En effet, $\mathcal{I}$ est similaire à $\mathcal{U}$, et $\mathcal{U}'$, après une première phase linéaire invoque $\mathcal{U}$ une fois si il n'y a pas de motifs-ou, ou autant de fois qu'il y a d'alternatives de motifs-ou à expanser. En oubliant les motifs-ou et en supposant que $\mathcal{U}$ est linéaire, on peut s'attendre au mieux à un comportement quadratique en la taille des filtrages, que l'on assimilera à la taille du fichier source.

Mais, même sans tenir compte des motifs-ou, le problème de la clause utile est NP-complet [6]. Il n'est donc pas très surprenant que l'algorithme $\mathcal{U}$ puisse prendre un temps exponentiel. Le problème apparaît à l'étape 2-(b) (section 3.1), les lignes de la matrice P qui commencent par un attrape-tout se retrouvent dans toutes les matrices $\mathcal{S}(c_k, P)$ (et éventuellement dans la matrice $\mathcal{D}(P)$). Le cas le plus défavorable est celui où $\vec{q}$ comprend des attrape-tout, P des constructeurs et des attrape-tout et où $\mathcal{U}(P, \vec{q})$ se révèle faux.

Le comportement exponentiel de $\mathcal{U}$ peut amener des temps de compilation prohibitifs pour des programmes de taille raisonnable, voire faible. Heureusement cela n'arrive pas très souvent en pratique. Toutefois, lorsqu'un tel programme est signalé, j'ai pu jusqu'ici comprendre pourquoi le temps calcul explosait et réagir par une *précaution* qui réglait le problème du programme signalé.

6.2. Précautions

Il convient de tenir compte du style des motifs qui proviennent de programmes écrits par des êtres humains. Les programmes bien écrits ne contiennent pas de clauses inutiles (cas favorable) et leurs filtrages sont exhaustifs (cas défavorable). Quant aux programmes mal écrits, il contiennent des clauses inutiles (cas défavorable) et des filtrages non-exhaustifs (cas favorable). On ne peut donc pas tout à fait ignorer le cas défavorable. Il vaut mieux se demander comment un être humain écrit un filtrage exhaustif ou une clause inutile. Or, une façon simple et répandue de rendre un filtrage exhaustif est d'ajouter un attrape-tout à la fin, tandis qu'une clause inutile porte un motif q assez précis, alors qu'il existe un motif plus prioritaire et moins précis. Ce dernier cas n'est pas préoccupant en lui même, car un motif q précis contient plutôt des constructeurs qui seront éliminés par la première phase de $\mathcal{U}'$, ce qui nous ramène au cas précédent. Les cas délicats se posent donc lorsque $\vec{q}$ est un vecteur d'attrape-tout et que P est de taille conséquente et contient une ligne peu précise, jusqu'au point de ne contenir que les attrape-tout.

Heureusement, il existe une solution pour diminuer radicalement le coût de $\mathcal{U}$ dans ce cas. En effet, si la matrice P comprend deux lignes distinctes $\vec{p}^{\,i}$ et $\vec{p}^{\,j}$ avec $\vec{p}^{\,i} \preceq \vec{p}^{\,j}$, alors on peut, par définition même de la relation de filtrage entre motif et valeur, enlever la ligne $\vec{p}^{\,j}$ de P sans rien changer aux instances de P. Or, un vecteur d'attrape-tout est moins précis que n'importe quel autre vecteur. Dès lors, toute matrice qui contient une telle ligne peut être remplacée par cette seule ligne.

Plus généralement les motifs des filtrages écrits en pratique sont souvent moins précis que certains des motifs qui les précèdent. On aura par exemple :

```
let f = function
| (1,2) -> 1
| (1,_) -> 2
| (_,2) -> 3
```

Lors de la recherche d'exhaustivité on peut ici supprimer le premier motif, plus précis que les deux suivants.

On est donc tenté, avant de lancer $\mathcal{U}$, de *minimiser* l'argument P, c'est-à-dire de remplacer P par la matrice formée des lignes de P qui sont minimales selon l'ordre de précision. Malheureusement, la minimisation n'est pas gratuite, puisqu'il faut au pire comparer deux à deux toutes les lignes de P.

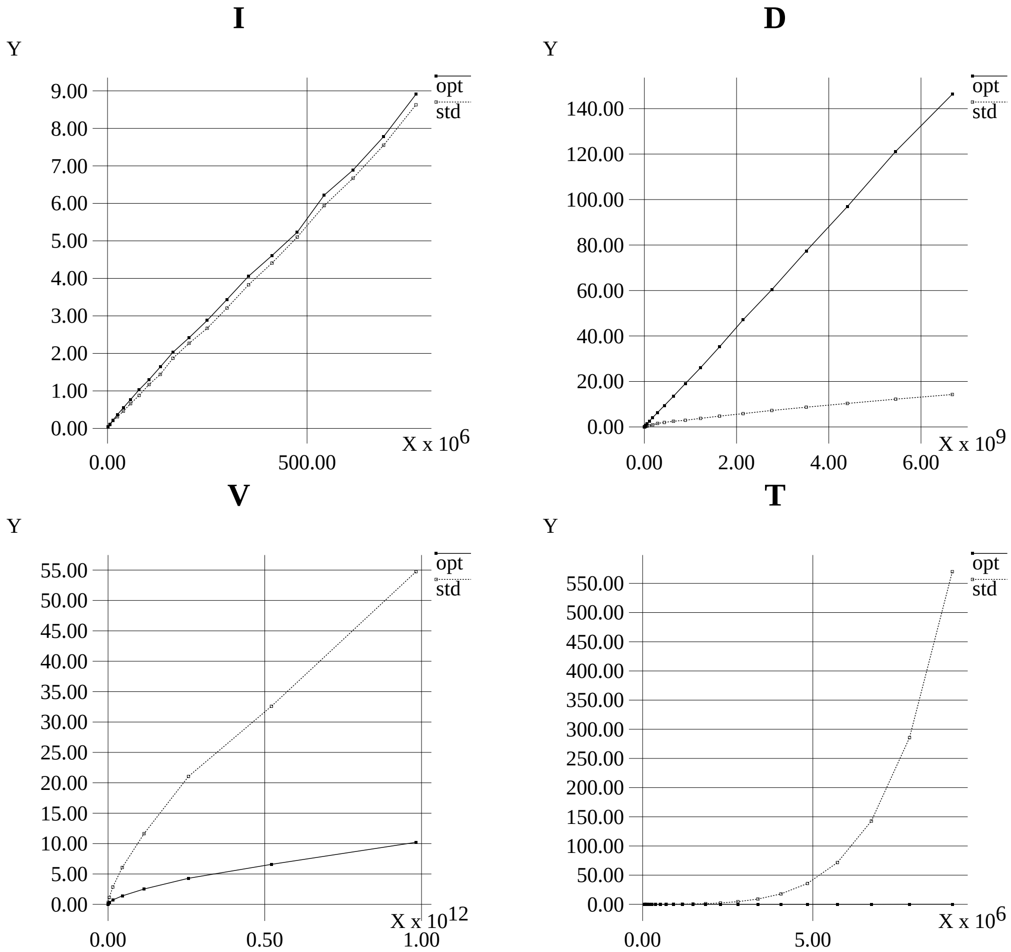
On peut simplifier P d'une autre façon. Deux motifs sont dit incompatibles quand ils n'ont pas d'instances en commun et compatibles dans le cas contraire. On décide la compatibilité par un algorithme récursif simple, voir par exemple [3]. Soit $\vec{p}^{\,i}$ une ligne de P incompatible avec $\vec{q}$. Alors, on peut enlever $\vec{p}^{\,i}$ de P puisqu'une instance de $\vec{q}$ n'est certainement pas une instance de $\vec{p}^{\,i}$. En fait les les lignes incompatibles avec $\vec{q}$ sont progressivement supprimés de P lors de l'étape 2-(a). Cependant, il est intéressant de les éliminer avant de procéder à la minimisation de la matrice P.

Une autre précaution qui se révèle moins discutable car de coût nul est, toujours lors de l'étape 2-(b), de procéder en premier à l'appel récursif sur la matrice par défaut. L'idée est ici qu'une instance de $\vec{q}$ qui n'est pas une instance de P, a de bonnes chances de porter en tête de sa première composante un constructeur absent dans les motifs plus prioritaires. Ce pari est souvent gagné dans le cas d'un motif attrape-tout réellement utile, ou ce qui revient au même dans le cas d'un filtrage non-exhaustif par oubli d'un constructeur.

6.3. Mesures

Les précautions prises sont d'ordre assez général, c'est-à-dire qu'elles ne traitent pas un exemple particulier, mais s'efforcent de traiter toute une « classe » d'exemples. Une fois un problème identifié, on peut concevoir une suite de filtrages de taille croissante correspondante et procéder à des mesures. On peut aussi concevoir de telles suites *a priori*.

FIG. 2 – Effet de la minimisation dans quatre expériences.



Sans prétendre à la complétude, je me suis livré à quelques expériences utilisant quatre schémas. Le schéma **I** est un filtrage des n premiers entiers, le schéma **D** est le filtrage entièrement composé d'attrape-tout, sauf la diagonale composée du constructeur `A` d'un type `t` à deux constructeur `A` et `B`. Le schéma **V** est le schéma qui déclenche le comportement exponentiel de la compilation du filtrage vers les arbres de décision (voir [4]). Enfin le schéma **T** est le mélange de deux matrices triangulaires dont la i-ème ligne commence par i attrape-tout et se poursuit par des `A` pour la première matrice et par `B` pour la seconde. Ce schéma déclenche le comportement exponentiel de $\mathcal{U}$ qui procède de la gauche vers la droite.

On mesure le temps pris par les vérifications du filtrage par différence du temps de compilation avec et sans les vérification de filtrage. Les expériences ont été faites sur un PC 1GHz et le temps, mesuré en secondes, est le temps effectivement pris et non le temps machine. Les courbes de la figure 2 montrent le temps pris par les vérifications en fonction du carré de la taille des fichiers compilés, dans deux cas : avec minimisation des matrices (**opt**) et sans (**std**). Les courbes montrent donc le caractère approximativement quadratique en pratique des vérifications. La courbe **std** de l'exemple **T** dénonce un comportement exponentiel.

Les expériences me conduisent à retenir la précaution de minimisation, surtout parce qu'elle permet d'éviter un comportement exponentiel dans l'expérience **T**. En outre, la minimisation n'a pratiquement pas d'impact dans l'expérience **I** (qui correspond à un schéma proche d'un cas fréquent pratique) et a un impact favorable dans l'expérience **V**. Dans le cas défavorable à la minimisation (expérience **D**), la minute de temps est atteinte pour une entrée de 160 lignes et colonnes. On peut donc considérer cet exemple comme assez éloigné d'un cas pratique et tolérer un facteur de ralentissement de environ 9 par rapport à la vérification sans minimisation.

7. Comparaisons et conclusion

Il n'y a, à ma connaissance, aucune littérature sur les vérifications du filtrage. Je me limite donc à l'examen rapide de deux compilateurs. Le compilateur Standard ML of New Jersey propose naturellement les vérifications puisqu'il procède à la compilation du filtrage vers les arbres de décision, toutefois il signale les motifs inutiles comme des clauses inutiles et non pas spécifiquement. Pour ce qui est du filtrage, Haskell est tout à fait comparable à ML, sans les motifs-ou. Le rapport Haskell [2] ignore la question des avertissements, mais le compilateur Glasgow Haskell qui utilise la technique de compilation vers les automates avec retour en arrière [1, 7] semble produire de bons avertissements. La vérification d'exhaustivité produit une liste d'exemples de motifs non-filtrés. L'examen du code du compilateur révèle un module spécifiquement chargé de détecter les anomalies du filtrage. Ce vérificateur est assez différent du nôtre, puisqu'il procède en une seule passe suivant de près l'algorithme de compilation vers les arbres de décision. Dès lors, un comportement exponentiel sera difficile à maîtriser. Or, ce comportement indésirable se produit bien au moins dans le cas des schémas `T` et `V`.

Il me semble que la question des avertissements du filtrage est, à des degrés divers, un peu négligée. Cette négligence est sans importance dans un prototype mais dommageable dans un compilateur qui ambitionne d'être diffusé, car les avertissements du filtrages sont très utiles, tant aux programmeurs débutants que chevronnés. Cet article montre que les anomalies peuvent s'étudier indépendamment des techniques de compilation du filtrage et qu'un intérêt porté aux anomalies pour elles-mêmes et un algorithme spécifique produisent au final les avertissements plus exploitables par le programmeur. Par ailleurs, le coût des vérifications en terme de temps de compilation semble bien maîtrisable en pratique.

Références

[1] Augustsson, L. Compiling pattern matching. In *Functional Programming Languages and Computer Architecture* (1985), J.-P. Jouannaud, Ed., Springer-Verlag, pp. 368–381. Lecture Notes in Computer Science 201.

[2] Hudak, P., Peyton Jones, S. L. *et al.* Haskell 98, a non-strict, purely functionnal language. `http://www.haskell.org/onlinereport/`.

[3] Le Fessant, F., and Maranget, L. Optimizing pattern-matching. In *International Conference on Functional Programming* (2001), ACM press.

[4] Maranget, L. Compiling lazy patern matching. In *Lisp and Functional Programming* (1992), ACM Press.

[5] Pettersson, M. A term pattern-match compiler inspired by finite automata theory. In *4th Workshop on Compiler Construction* (1992), Springer-Verlag. Lecture Notes in Computer Science 641.

[6] Sekar, R. C., Ramesh, R., and Ramakrishnan, I. V. Adaptive pattern matching. In *International Colloquium on Automata Languages and Programming* (1992), Springer-Verlag. Lecture Notes in Computer Science 623.

[7] Wadler, P. Efficient compilation of pattern matching. In *Peyton Jones S. L. , The Implementation of Functional Programming Languages*. Prentice-Hall, 1987, ch. 6.