Algorithmes et complexités de la réduction statique minimale

Ludovic Henrio & Bernard Paul Serpette & Szabolcs Szentes

Inria Sophia-Antipolis
2004 route des Lucioles – B.P. 93
F-06902 Sophia-Antipolis, Cedex
First.Last@inria.fr
http://www.inria.fr/oasis/First.Last

Résumé

La réduction statique minimale (*RSM*) est l'analyse de flot de contrôle d'ordre 0 (*0CFA*). Cet article présente une série d'algorithmes résolvant cette analyse ainsi que leurs complexités respectives. Même si cette analyse est souvent annoncée comme étant calculable en un temps cubique par rapport à la taille du programe étudié, un algorithme complet atteignant cette complexité n'est pas si évident. Avec le λ-calcul comme langage de description des programmes, nous commençons par un algorithme naïf et, pas à pas, nous aboutissons à celui qui s'exécute, dans le pire des cas, en un temps cubique. Nous donnerons les exemples qui prouvent que ces complexités maximales peuvent être atteintes.

Cet article a été soumis en anglais, la version originale se trouve en tant que rapport de recherche INRIA.

1. Introduction

L'analyse décrite dans cet article définit, pour chaque nœud de l'arbre de syntaxe abstraite (AST), un ensemble de valeurs que ce nœud pourrait calculer à l'exécution. Cette analyse a tout d'abord été étudiée dans le cadre de l'interprétation abstraite : [18, 19, 20, 16, 2, 17, 11, 4, 1]. Puis, Nevin Heintze entama une étude basée sur la résolution de systèmes de contraintes (inéquations ensemblistes) : [5, 14, 9, 8, 10, 6, 13]. Plus récemment, cette analyse fut étudiée du point de vue de la théorie des types : [15, 7, 12, 21].

Dans ce papier, afin de pouvoir analyser la complexité, nous essayons d'expliciter, aussi loin que possible, l'algorithme de résolution. En d'autres termes, nous intégrons dans les algorithmes ce qu'un interprète abstrait, un résolveur de contraintes ou un vérificateur de type étaient censés calculer.

Sans considération de la technologie utilisée pour résoudre cette analyse, il existe une grande variété de stratégie concernant l'adéquation entre la finesse des résultats et la complexité de la résolution. Par exemple, la réduction statique (SRA [16]) va d'une analyse exacte imitant l'évaluateur à une analyse minimale (RSM) en relation directe avec l'analyse de flot de contrôle d'ordre 0 (OCFA [19]) et largement assimilable à la *Set Based Analysis* (SBA [5]).

Puisque nous désirons comparer différents algorithmes, il est nécessaire d'utiliser une spécification claire, nette et précise de l'analyse. C'est pourquoi nous avons choisi la réduction statique minimale pour notre étude : pour le λ-calcul, la RSM est définie comme la plus petite fonction satisfaisant un jeu de 3 règles extrêmement simples.

La contribution majeure de ce papier est de proposer des algorithmes *prêt à l'emploi* pour des *langages ouverts*. Prêt à l'emploi dans le sens où les seules opérations non définies sont $\forall x \in S : P$ (pour tout élément x de S, effectuer P) qui est un itérateur dépendant du type choisi pour S (**map**, **for** ...) et les fonctions sur les ensembles ($\in, \subset, \cup$).

Le langage des programmes analysés par les algorithmes est *ouvert* dans le sens où l'on n'impose pas que les programmes aient subit, au préalable, une transformation; notamment une transformation CPS (*Continuation Passing Style*) que l'on retrouve souvent comme prérequise. Nous reviendrons succinctement sur cette question à la fin de l'article.

L'article est organisé comme suit. La section 2 présente le contexte et la RSM. La section 3 explique quatre algorithmes différents calculant la RSM et détermine leurs complexités. La section 4 révèle les pires cas des algorihmes précédent en présentant plusieurs séries de programmes *assassins*. La section 5 fait quelques remarques sur la spécification de la 0CFA et montre certaines directions possibles de travaux futurs.

La preuve Coq de la correction de la RSM, la définition Bigloo des quatres algorithmes et des programmes *assassins* se trouvent dans l'archive :
`ftp://ftp-sop.inria.fr/oasis/Bernard.Serpette/jfla04.tar.gz`

2. Définitions

Cette section définit ce que les algorithmes décrits dans ce papier sont censés résoudre. Tout d'abord nous donnons le langage analysé par les algorithmes. Pour se restreindre à l'essentiel, ce langage est le λ-calcul, dans [3], l'analyse est étendue pour les langages orientés à objets et possédant des traits impératifs. Puis, nous abordons brièvement la spécification générique de la réduction statique (SRA). La réduction statique minimale (RSM) est définie comme une instantiation particulière de la SRA. Enfin, nous apportons quelques équations générales concernant la taille des programmes.

2.1. Le langage d'entrée

Le langage des programmes à analyser, le λ-calcul, est décrit par les définitions Coq suivantes :

Variable var	:	**Set**.
Inductive *Expr*	:	**Set** :=
Var	:	var $\rightarrow$ *Expr*
| *Lambda*	:	var $\rightarrow$ *Expr* $\rightarrow$ *Expr*
| *App*	:	*Expr* $\rightarrow$ *Expr* $\rightarrow$ *Expr*

L'ensemble *Expr* est paramétrisé par un ensemble de variables nommé *var*. On suppose l'existence d'une égalité entre ces variables. Les fonctions, éléments de *Lambda*, sont notés $\lambda x.B$ et les nœuds applications, éléments de *App*, sont notés $(F\ A)$. Pour ce simplissime langage, les valeurs sont les fonctions : $Value = Lambda$.

2.2. SRA générique

La SRA générique est paramétrisée par une **fonction d'approximation** dont la signature est $App, Lambda, Value \rightarrow Lambda$. Cette fonction décide, pour chaque cas particulier d'application, de la fonction qu'il convient d'analyser. Pour être valide, la fonction d'approximation $\mathcal{F}$ doit retourner une fonction α-équivalente (égalité modulo renommage des variables liées) à son deuxième argument : $\mathcal{F}(e, l, a) \equiv_\alpha l$. Pour toute fonction d'approximation $\mathcal{F}$, la réduction statique est définie comme la **plus petite** relation, des expressions vers les valeurs ($Expr \rightarrow Value$), satisfaisant les règles suivantes :

$$\frac{l \in Lambda}{l \rightarrow l}$$

$$\frac{F \to f \ \wedge \ A \to a \ \wedge \ \lambda x.B = \mathcal{F}((F\ A), f, a)}{x \to a}$$

$$\frac{F \to f \ \wedge \ A \to a \ \wedge \ \lambda x.B = \mathcal{F}((F\ A), f, a) \ \wedge \ B \to v}{(F\ A) \to v}$$

Le point fort des fonctions d'approximation est qu'elles ont la capacité, en respectant la limitation d'α-équivalence, de *créer* de nouvelles expressions. Par exemple on peut avoir $\mathcal{F}(e, \lambda x.x, a) = \lambda y.y$, où la variable y n'apparaît dans le programme principal, et ainsi éviter à la variable x d'avoir a comme valeur possible. Inversement, une même variable x peut être introduite par deux fonctions différentes, par exemple $\lambda x.x$ et $\lambda x.(x\ x)$, mais on appauvrit inutilement les valeurs potentielles de x. L'analyse calculera de meilleures résultats si l'on remplace la fonction $\lambda x.x$ par $\lambda y.y$ où la variable y n'apparaît pas dans le programme principal. Une *alpha*-conversion systématique est généralement faite à la construction de l'AST.

Si la fonction d'approximation décide de créer systématiquement des nouvelles expressions, on imite alors le comportement de l'évaluateur, mais dans ce cas, l'analyse n'a pas de critère de terminaison. Avec cette fonction d'approximation, les expressions ont au plus un successeur par la relation $\to$ qui peut être donc vu comme une fonction partielle.

La réduction statique peut être aussi vue comme une *fonction* $\mathcal{S}$ des expressions vers des *ensembles* de valeurs ($Expr \to \mathcal{P}(Value)$) avec, par définition : $v \in \mathcal{S}(e) \Leftrightarrow (e \to v)$. $\mathcal{S}$ dénote les successeurs du graphe induit par la relation $\to$. Ainsi, $\mathcal{S}$ est la plus petite fonction satisfaisant les règles suivantes :

$$\frac{l \in Lambda}{l \in \mathcal{S}(l)}$$

$$\frac{f \in \mathcal{S}(F) \ \wedge \ a \in \mathcal{S}(A) \ \wedge \ \lambda x.B = \mathcal{F}((F\ A), f, a)}{a \in \mathcal{S}(x)}$$

$$\frac{f \in \mathcal{S}(F) \ \wedge \ a \in \mathcal{S}(A) \ \wedge \ \lambda x.B = \mathcal{F}((F\ A), f, a) \ \wedge \ v \in \mathcal{S}(B)}{v \in \mathcal{S}((F\ A))}$$

Les fonctions d'approximation sont dans un treillis dont l'élément minimal est $\mathcal{F}_\perp(e, l, a) = l$. Cette fonction définit la réduction statique minimale (RSM) qui est analysée tout au long de ce papier.

2.3. Réduction statique minimale

Avec cette fonction d'approximation $\mathcal{F}_\perp(e, l, a) = l$, on observe que l'analyse ne manipule que les sous-expressions du programme principal. Cet ensemble fini, dénoté par $\mathcal{E}$, est connu dès le début de l'analyse. Il peut se partitionner par projection sur Var, $Lambda$ et App : $\mathcal{E} = \mathcal{V} \oplus \mathcal{L} \oplus \mathcal{A}$. Le résultat de la RSM est donc un graphe de $\mathcal{E}$ dans $\mathcal{L}$ et ainsi, $\mathcal{S}$ devient une fonction de $\mathcal{E}$ vers $\mathcal{P}(\mathcal{L})$. Suivant ces considérations, les règles pour la RSM se simplifient et deviennent :

$$\frac{l \in \mathcal{L}}{l \in \mathcal{S}(l)} \tag{1}$$

$$\frac{(F\ A) \in \mathcal{A} \ \wedge \ \lambda x.B \in \mathcal{S}(F)}{\mathcal{S}(A) \subseteq \mathcal{S}(x)} \tag{2}$$

$$\frac{(F\ A) \in \mathcal{A} \ \wedge \ \lambda x.B \in \mathcal{S}(F)}{\mathcal{S}(B) \subseteq \mathcal{S}((F\ A))} \tag{3}$$

Ce qui doit se lire comme : (1) la valeur d'une fonction est elle-même ; si l'on connait une des fonctions appliquées sur un nœud d'application, (2) il faut assurer que les valeurs de l'argument soit stockées dans la paramètre formel et (3) ramener comme résultat de l'application les valeurs produites par le corps de la fonction.

2.4. Taille et position

Toutes les complexités données dans cet article sont relatives à la *taille* N du programme principal. Nous définissons cette taille comme étant le cardinal de l'ensemble $\mathcal{E}$: $N = \|\mathcal{E}\|$. Ceci n'est pas forcément la manière usuelle de compter les sous-expressions d'une expression : en général chaque utilisation d'une variable est prise en compte. Dans notre cas, la taille de l'expression $\lambda x.(x\ x)$ est 3 : une fonction, une application et **une** variable. Cette approche est naturelle car nous cherchons à construire un graphe comportant N nœuds. Pour la RSM, toutes les occurences d'une même variable ont les mêmes valeurs, quelques soient leurs positions dans l'AST (ce n'est plus forcement le cas pour les langages impératifs et l'on parle dans ce cas d'une analyse *flow sentitive*).

Ainsi, l'*arbre* de syntaxe abstraite n'est plus réellement un arbre puisque toutes les instances d'une même variable, autant les utilisations que le paramètre formel, sont unifiés dans le même nœud. Nous nous interesserons plus particulièrement au sous-graphe de l'AST correspondant à la notion de **sous-expression**, c'est à dire au graphe de l'AST auquel les arcs reliant les fonctions à leurs paramètres formels ont été supprimés. Ceci peut se formaliser par la fonction $SubExpr$:

$$\begin{array}{lll}
\lambda x.B \in \mathcal{L} & \Rightarrow & SubExpr(\lambda x.B) = \{B\} \\
(F\ A) \in \mathcal{A} & \Rightarrow & SubExpr((F\ A)) = \{F,\ A\} \\
x \in \mathcal{V} & \Rightarrow & SubExpr(x) = \emptyset
\end{array}$$

De la même manière nous définissons la relation inverse *"est une sous-expression de"* par la plus petite fonction Pos satisfaisant :

$$\begin{array}{lll}
\lambda x.B \in \mathcal{L} & \Rightarrow & \lambda x.B \in Pos(B) \\
(F\ A) \in \mathcal{A} & \Rightarrow & (F\ A) \in Pos(F) \\
(F\ A) \in \mathcal{A} & \Rightarrow & (F\ A) \in Pos(A)
\end{array}$$

Dans le λ-calcul, une sous-expression X est soit en position fonctionnelle d'une application (i.e. $(X\ ?)$), soit en position d'argument d'une application (i.e. $(?\ X)$), soit directement le corps d'une fonction (i.e. $(\lambda ?.X)$).

En suivant les projections des expressions sur les variables, les fonctions et les applications nous avons : $\|\mathcal{E}\| = N = \|\mathcal{V} \oplus \mathcal{L} \oplus \mathcal{A}\| = \|\mathcal{V}\| + \|\mathcal{L}\| + \|\mathcal{A}\| = N_{\mathcal{V}} + N_{\mathcal{L}} + N_{\mathcal{A}}$. Puisque les valeurs sont les fonctions nous avons : $\forall e \in \mathcal{E} : \|\mathcal{S}(e)\| \leq N_{\mathcal{L}}$. Puis, sachant qu'un graphe et son inverse a le même nombre d'arcs, nous avons : $\sum_{e\in\mathcal{E}} \|Pos(e)\| = \sum_{e\in\mathcal{E}} \|SubExpr(e)\| \leq N_{\mathcal{L}} + 2N_{\mathcal{A}}$. L'inégalité est du au cas particulier : $SubExpr((x\ x)) = \{x\}$.

On peut donc conlure par cette inéquation qui sera utilisé dans les prochains calculs de complexité [1] :

$$\sum_{e\in\mathcal{E}} \|\mathcal{S}(e)\| \times \|Pos(e)\| \leq \sum_{e\in\mathcal{E}} N_{\mathcal{L}} \times \|Pos(e)\| = N_{\mathcal{L}} \times \sum_{e\in\mathcal{E}} \|Pos(e)\| \leq N_{\mathcal{L}}(N_{\mathcal{L}} + 2N_{\mathcal{A}}) \leq 2N^2$$

3. Algorithmes

Cette section décrit quatre algorithmes calculant la réduction statique minimale. Le *graphe* obtenu par ces algorithmes possède N *nœuds* et potentiellement $\mathcal{O}(N^2)$ *arcs* reliant les nœuds à des *valeurs* (i.e. $Value = Lambda$).

3.1. Algorithme naïf

Le premier et le plus simple des algorithmes est une résolution directe des règles de la RSM. En premier, il applique la règle 1 sur toutes les fonctions. Puis, pour chaque nœud d'application et

[1] Avec une analyse plus fine, cette borne supérieur peut se ramener à : $2N_{\mathcal{L}}(N_{\mathcal{A}} + 1)$

0:**0**	$\forall l \in \mathcal{L} : \ \mathcal{S}(l) \leftarrow \{l\}$	*Règle 1*
1:**0**	$more \leftarrow$ **true**	*Initialisation du point fixe*
2:**2**	**while**($more$)	*Itération du point fixe*
3:**2**	$\quad more \leftarrow$ **false**	*Réinitialisation*
4:**2**	$\quad \forall (F\ A) \in \mathcal{A} :$	*Pour chaque application*
5:**3**	$\quad\quad \forall \lambda x.B \in \mathcal{S}(F) :$	*Pour chaque fonction appliquée*
6:**4**	$\quad\quad\quad Update(x, A)$	*Vérifie la règle 2*
7:**4**	$\quad\quad\quad Update((F\ A), B)$	*Vérifie la règle 3*
8:**4**	$Update(dst, src)$	
9:**4**	$\quad$ **if** $(\mathcal{S}(src) \subseteq \mathcal{S}(dst))$ **return**	*La règle est déjà satisfaite*
10:**2**	$\quad \mathcal{S}(dst) = \mathcal{S}(dst) \cup \mathcal{S}(src)$	*Mets à jour $\mathcal{S}$ pour satisfaire la règle*
11:**2**	$\quad more \leftarrow$ **true**	*Au moins un arc est ajouté*

Figure 1: Algorithme naïf

chaque fonction satisfaisant les prémisses des règles 2 et 3, il vérifie les conclusions correspondantes et rajoute un arc si cela est nécessaire. Un point fixe est atteint lorsque plus aucun arc ne peut être ajouté. Cet algorithme est décrit dans la figure 1. Chaque ligne est préfixée par *n*:**k** où n est un numéro de ligne et où **k** est le degré maximum du nombre de fois que cette ligne peut être exécutée. (i.e. *n*:**k** s'interprète comme : la ligne n peut être exécutée, dans le pire des cas, $\mathcal{O}(N^k)$ fois).

Dans le pire des cas, le graphe calculé est *dense* : le graphe possède $\mathcal{O}(N^2)$ arc, ou $\sum_{e \in \mathcal{E}} \|\mathcal{S}(e)\| = \mathcal{O}(N^2)$. Un programme pernicieux exigera qu'un nombre fini d'arcs soient ajoutés au graphe à chaque itération, tout en imposant que le graphe final soit dense. Un tel programme nécessite que le nombre d'itération pour atteindre le point fixe soit proportionnel à N^2. On peut en déduire que les lignes *2*, *3*, *4*, *10* et *11* peuvent être exécutées $\mathcal{O}(N^2)$ fois. Si le nombre de nœuds d'application est du même ordre que N ($N_{\mathcal{A}} = \mathcal{O}(N)$), alors la ligne *5* est exécutée $\mathcal{O}(N^3)$ fois. Si le sous-graphe induit par $\mathcal{A}$ est, lui aussi, dense ($\sum_{(F\ A) \in \mathcal{A}} \|\mathcal{S}(F)\| = \mathcal{O}(N^2)$) [2] les lignes *6* et *7* sont exécutées $\mathcal{O}(N^4)$ fois. Donc la fonction *Update* peut être appelée $\mathcal{O}(N^4)$ fois. Puisque le temps de calcul pour le test d'inclusion sur les ensembles est au pire linéaire, la complexité générale de l'algorithme est en $\mathcal{O}(N^5)$ [3]. Les programmes *assassins* décrits dans la section 4 montrent que cette complexité peut est atteignable.

Les principaux défauts de cet algorithme naïf sont :

1. Les nœuds d'application sont choisis dans un ordre prédéfini, i.e. dans l'ordre dans lequel on les a placé dans $\mathcal{A}$. Il serait certainement profitable de visiter les nœuds d'application des fonctions appelées avant ceux des fonctions appelantes. Ce sera l'optimisation proposée par l'algorithme *influencé par l'évaluateur*.

2. Entre chaque itération l'algorithme ne tient pas compte des nœuds sur lesquels la procedure *Update* a déjà été invoquée. Il est facile de se rendre compte que si $Update(dst, src)$ est appelée deux fois sur le même nœud dst, et si aucune valeur n'a été ajoutée à $\mathcal{S}(src)$ entre ces deux appels, alors le deuxième appel devient inutile puisque l'on peut aisément prouver que l'inclusion $\mathcal{S}(src) \subseteq \mathcal{S}(dst)$ reste vrai. Cette constatation sera utilisée par l'algorithme manipulant des *valeurs triées*.

3. Lorsqu'un arc est rajouté, l'algorithme n'essaye pas d'extraire les nœuds d'application pouvant être influencés par cette ajout. L'algorithme *orienté par la position* utilisera ce fait.

[2] Ce qui s'obtient lorsque pour $\mathcal{O}(N)$ nœuds d'application $(F\ A)$ on a $\|\mathcal{S}(F)\| = \mathcal{O}(N)$

[3] La complexité réelle est $\mathcal{O}((N_{\mathcal{V}} + N_{\mathcal{A}}) N_{\mathcal{L}} N_{\mathcal{A}} N_{\mathcal{L}} N_{\mathcal{L}})$

0:**0**	$\forall l \in \mathcal{L} : \mathcal{S}(l) \leftarrow \{l\}$	*Règle 1*
1:**0**	$time \leftarrow 0$	*Initialisation du temps global*
2:**0**	$\forall l \in \mathcal{L} : Time(l) \leftarrow -1$	*Initialisation du temps associé aux fonctions*
3:**0**	$more \leftarrow$ **true**	*Initialisation du point fixe*
4:**2**	**while**$(more)$	*Itération du point fixe*
5:**2**	$more \leftarrow$ **false**	*Réinitialisation*
6:**2**	$Eval(Main)$	*Évaluation abstraite du programme principal*
7:**2**	$time \leftarrow time + 1$	*Incrément du temps*
8:**3**	$Eval(e)$	*Évaluation abstraite de e*
9:**3**	**if** $(e = (F\ A) \in \mathcal{A})$	*Seuls les nœuds d'application sont intéressants*
10:**3**	$Eval(F)$	*Évaluation de la partie fonctionnelle*
11:**3**	$Eval(A)$	*Évaluation de l'argument*
12:**3**	$\forall f = \lambda x.B \in \mathcal{S}(F) :$	*Pour chaque fonction*
13:**4**	$Update(x, A)$	*Vérification de la règle 2*
14:**4**	**if** $Time(f) \neq time$	*On analyse les corps qu'une seule fois par itération*
15:**3**	$Time(f) \leftarrow time$	*Mets à jour le temps pour cette fonction*
16:**3**	$Eval(B)$	*Évaluation du corps*
17:**4**	$Update(e, B)$	*Vérification de la règle 3*

Figure 2: Algorithme influencé par l'évaluateur

3.2. Algorithme influencé par l'évaluateur

Nous verrons dans la section 4 que, pour l'algorithme naïf, le nombre d'itération nécessaire pour atteindre le point fixe est dépendant de l'ordre dans lequel on prend les nœuds d'application : on peut s'attendre à de meilleurs résultats si ces nœuds d'application sont pris dans un ordre plus approprié. Intuitivement, en suivant le chemin pris par un évaluateur, il semblerait meilleur de suivre le corps d'une fonction appelée avant d'analyser le nœud d'application où cette fonction est activée. En cas de récursion, et pour assurer la terminaison de l'algorithme, on devra, d'une manière ou d'une autre, s'arrêter dans cette analyse en profondeur et ainsi conserver une résolution de point fixe. Cet algorithme influencé par l'évaluateur est donné dans la figure 2. A chaque itération, la variable *time* est incrémenté et chaque fonction possède sa propre annotation de temps (la table d'association $Time$ des fonctions vers les entiers); $Time(f) = n$ montre que le corps de la fonction f a été analysé à la n^{ieme} itération. La procedure *Update* est la même que celle définie pour l'agorithme naïf (Figure 1).

On peut dire que cet algorithme a été popularisé par la thèse d'Olin Shivers. Cette algorithme a beaucoup de variantes, nous discuterons de l'une d'entre elles dans la section 5. On notera que cet algorithme ne remplit pleinement le contrat de la RSM que si le programme principal ne contient pas de code mort : le corps d'une fonction n'est analysé que si elle est appliquée.

L'analyse de complexité faite pour l'algorithme naïf se reporte sur cet algorithme influencé par l'évaluateur. Si $\mathcal{O}(N^2)$ itérations sont nécessaires pour atteindre le point fixe (lignes *4* à *7*), en supposant l'absence de code mort, la procedure *Eval* est appelée $\mathcal{O}(N^3)$ fois (pour chaque itération, une fois pour $\mathcal{O}(N)$ nœuds). Si $\sum_{(F\ A)\in\mathcal{A}} \|\mathcal{S}(F)\| = \mathcal{O}(N^2)$, alors la procédure *Update* est appelée $\mathcal{O}(N^4)$ fois et donc cet algorithme possède aussi, dans le pire des cas, une complexité en $\mathcal{O}(N^5)$.

Ce résultat de $\mathcal{O}(N^5)$ est une borne supérieure de la compléxité mais, contrairement à l'algorithme naïf, le programme *assassin* atteignant cette compléxité n'a pas été immédiat à trouver.

0:**0**	$\forall l \in \mathcal{L}: \ \mathcal{S}(l) \leftarrow (< l, 0 >)$	*Règle 1*
1:**0**	$time \leftarrow 0$	*Initialisation du temps*
2:**0**	$more \leftarrow$ **true**	*Initialisation du point fixe*
3:**2**	**while**$(more)$	*Itération du point fixe*
4:**2**	$\quad more \leftarrow$ **false**	*Réinitialisation*
5:**2**	$\quad \forall (F\ A) \in \mathcal{A}:$	*Pour chaque application*
6:**3**	$\qquad \forall < \lambda x.B, t > \in \mathcal{S}(F):$	*Pour chaque fonction appliquée*
7:**4**	$\qquad\qquad Update(x, A, t \geq time)$	*Vérifie la règle 2*
8:**4**	$\qquad\qquad Update((F\ A), B, t \geq time)$	*Vérifie la règle 3*
9:**2**	$\quad time \leftarrow time + 1$	*Incrément du temps*
10:**4**	$Update(dst, src, new?)$	
11:**4**	$\quad$ **if** $new?$	*Si la fonction est nouvelle*
12:**2**	$\qquad \forall < v, \star > \in \mathcal{S}(src):$	*Vérifie toutes les valeurs de src*
13:**3**	$\qquad\quad UpdateValue(dst, v)$	*Mets à jour une valeur*
14:**4**	$\quad$ **else**	*Pour les fonctions plus anciennes*
15:**4**	$\qquad \forall < v, t' > \in \mathcal{S}(src)\ with\ t' \geq time:$	*Vérifie les nouvelles valeurs*
16:**3**	$\qquad\quad UpdateValue(dst, v)$	*Mets à jour une valeur*
17:**3**	$UpdateValue(dst, v)$	
18:**3**	$\quad$ **if** $(< v, \star > \in \mathcal{S}(dst))$ **return**	*La valeur est déjà présente*
19:**2**	$\quad \mathcal{S}(dst) = \{< v, time + 1 >\} \cup \mathcal{S}(dst)$	*Mets à jour $\mathcal{S}$*
20:**2**	$\quad more \leftarrow$ **true**	*Un arc a été ajouté!*

Figure 3: Algorithme avec des valeurs triées

3.3. Algorithme avec des valeurs triées

Afin d'optimiser le nombre d'appel à la procédure *Update* du précédent algorithme, on annote les arcs, à leurs créations, par le numéro de l'itération en cours, considéré comme un temps virtuel. Au temps T, on associe le temps $T + 1$ à chaque arc créé. Ainsi, au temps T, une valeur d'un nœud est considérée comme *neuve* si son temps est supérieur ou égal à T, et est appelée *ancienne* dans le cas contraire. On peut vérifier que l'analyse d'un nœud d'application est influencé par :

1. les nouvelles valeurs de l'argument. Ces valeurs doivent être propagées à **toutes** les fonctions appliquées : même les anciennes fonctions doivent être averties de ces nouveaux arguments.

2. les nouvelles valeurs fonctionnelles. **Toutes** les valeurs de l'argument doivent arriver au paramètre formel et **toutes** les valeurs du corps de la nouvelle fonction doivent remonter au nœud d'application.

3. toutes les valeurs des corps des fonctions appelées. Malheureusement, **toutes** les valeurs fonctionnelles doivent être visitées afin de propager les éventuelles nouvelles valeurs au nœud d'application.

Ceci permet d'éviter de considérer la propagation des anciennes valeurs de l'argument sur les anciennes fonctions appelées. Cet algorithme avec des valeurs triées est décrit dans la figure 3. La nouvelle procédure *Update* reçoit un booléen indiquant si la règle est vérifiée pour une *nouvelle* fonction. Dans ce cas, comme dans l'algorithme précédent, toutes les valeurs de la source sont visitées. Pour les anciennes valeurs de la partie fonctionnelle, seules les nouvelles valeurs de la source sont analysées.

Le point crucial pour calculer la complexité est d'observer que, lors de l'ajout d'une valeur v dans le nœud X au temps t, chaque position P de X peut induire au plus $2 * N$ appels à la procédure *UpdateValue* au temps $t + 1$. En suivant la description de l'algorithme, faisons une étude de cas sur cette position P.

1. $P = (F\ X)$. La nouvelle valeur v impose au plus N appels à *UpdateValue* au temps $t + 1$: un appel pour chaque fonction de F (au plus N).
2. $P = (X\ A)$. À l'itération suivante, deux nouveaux appels à la procédure *Update* seront la conséquence de cet ajout (ligne *7* et *8*), et donc, cette nouvelle valeur induiera moins de $2N$ appels à *UpdateValue* au temps $t + 1$.
3. $P = \lambda y.X$. La nouvelle valeur v causera au plus N appels à *UpdateValue* au temps $t + 1$. Un appel pour toutes les configurations (au plus N) où $\lambda y.X$ se retrouve comme valeur dans la position fonctionnelle d'un nœud d'application.

Si on note n_t^e le nombre de valeurs rajoutées au nœud e au temps t, u_t le nombre d'appel, au temps t, à la procédure *UpdateValue* , alors nous venons juste de prouver que : $u_{t+1} \leq 2N \sum_{e\in\mathcal{E}} n_t^e \times \|Pos(e)\|$. Si on note T le nombre d'itération pour atteindre le point fixe (i.e. la dernière valeur de la variable *time*), alors :

$$\begin{aligned}
\sum_{t\leq T} u_{t+1} &\leq \sum_{t\leq T} 2N \sum_{e\in\mathcal{E}} n_t^e \times \|Pos(e)\| \\
&= 2N \sum_{e\in\mathcal{E}} (\sum_{t\leq T} n_t^e) \times \|Pos(e)\| \\
&= 2N \sum_{e\in\mathcal{E}} \|\mathcal{S}(e)\| \times \|Pos(e)\| \\
&\leq 4N^3 \quad \text{(voir section 2.4)}
\end{aligned}$$

Le terme $\sum_{t\leq T} u_{t+1}$ représente le nombre de fois que l'algorithme passe par la ligne *17*, ce nombre est du même ordre pour les lignes *13*, *16* et *18*. Pour cette dernière ligne, nous supposons que le test d'appartenance dans un ensemble peut se faire en temps constant, ce qui est possible en utilisant des vecteurs de bits pour représenter les ensembles.

Connaissant le majorant pour le nombre d'arcs d'un graphe, on en déduit la complexité des lignes *19* et *20* qui ne peut excéder $\mathcal{O}(N^2)$. L'analyse de complexité de l'algorithme naïf dit que toutes les autres lignes ne peuvent pas dépasser $\mathcal{O}(N^4)$. Ceci nous permet de conclure que, dans le pire des cas, la complexité de l'agorithme avec des valeurs triées est en $\mathcal{O}(N^4)$.

Pour les lignes *12* et *15* nous donnons le résultat d'un programme *assassin* de la section 4. On note que la ligne *12* peut monter jusqu'à 4 (mais dans ce cas la plupart des $\mathcal{S}(src)$ sont vides) ou être 3 (les $\mathcal{S}(src)$ contiennent un nombre constant de valeurs). Cette considération est aussi valide pour la ligne *15* où un programme *assassin* atteint une complexité d'ordre 4, mais, avec une définition des ensembles utilisant des vecteurs de bits, cette ligne peut cacher une complexité d'ordre 5 (tout le vecteur doit être parcouru avant de s'apercevoir qu'il ne contient pas de nouvelles valeurs). On contourne ce problème en utilisant à la fois des vecteurs de bits et des listes triées comme représentation des ensembles de valeurs (i.e. la ligne *19* allume un bit dans le vecteur **et** rajoute la nouvelle valeur en tête d'une liste).

En contraste avec l'algorithme naïf, nous gagnons un ordre de magnitude dans la complexité en éliminant les appels à la procédure *Update* qui ont été déjà précédemment effectués. C'est seulement par le fait que $\mathcal{O}(N^2)$ itérations peuvent être exécutées pour atteindre le point fixe que cet algorithme est en $\mathcal{O}(N^4)$. Pour $\mathcal{O}(N)$ itérations, la complexité redescend à $\mathcal{O}(N^3)$.

3.4. Algorithme orienté par la position

L'analyse de complexité de l'algorithme avec des valeurs triées suggère que, lors de l'ajout d'une valeur dans un nœud, la position de ce nœud est cruciale. Cette considération amène à l'algorithme orienté par la position décrit dans la figure 4. Analysons les conséquences de l'ajout d'une nouvelle valeur $v = \lambda y.B$ dans le nœud X à la position P.

00:**0**	$\forall l \in \mathcal{L} :\ UpdateValue(l,l)$	*Entame la récursion par la règle 1*
01:**3**	$UpdateValue(dst, v)$	*Ajoute une valeur v dans le nœud dst*
02:**3**	**if** $(v \in \mathcal{S}(dst))$ **return**	*Vérifie si cela n'est pas déjà fait*
03:**2**	$\mathcal{S}(dst) \leftarrow \mathcal{S}(dst) \cup \{v\}$	*Rajoute cette valeur*
04:**2**	$\forall P \in Pos(dst) : UpdatePos(P, dst, v)$	*Mets à jour chaque position*
05:**2**	$UpdatePos(P, dst, v = \lambda x.B)$	*Mise à jour d'une position*
06:**2**	**if** $(P = (dst\ A) \in \mathcal{A})$	*Cas 1*
07:**2**	$Redexes(v) \leftarrow Redexes(v) \cup \{P\}$	*Mise à jour des redexes*
08:**2**	$\forall (a \in \mathcal{S}(A)) :\ UpdateValue(x, a)$	
09:**2**	$\forall (r \in \mathcal{S}(B)) :\ UpdateValue(P, r)$	
10:**2**	**if** $(P = (F\ dst) \in \mathcal{A})$	*Cas 2*
11:**2**	$\forall (\lambda y.C \in \mathcal{S}(F)) :\ UpdateValue(y, v)$	
12:**2**	**if** $(P = \lambda z.dst \in \mathcal{L})$	*Cas 3*
13:**2**	$\forall (E \in Redexes(P)) :\ UpdateValue(E, v)$	

Figure 4: Algorithme orienté par la position

1. Si $P = (X\ A)$, le paramètre formel y peut recevoir des nouvelles valeurs via A, et P peut recevoir des nouvelles valeurs par le corps B.
2. Si $P = (F\ X)$, tous les paramètres formels des valeurs de F peuvent recevoir cette nouvelle valeur v.
3. Si $P = \lambda z.X$, tous les nœuds d'application où cette fonction est appliquée peuvent recevoir cette nouvelle valeur v. Cet ensemble de nœuds d'application est construit incrémentalement dans la table d'association *Redexes* ($\mathcal{L} \rightarrow \mathcal{P}(\mathcal{A})$). Cette table est mise à jour chaque fois qu'une nouvelle valeur est introduite dans la partie fonctionnelle d'une application (ici dans le cas 1).

Cet algorithme est dans l'esprit de celui décrit dans la thèse d'Andrew Ayers [2]. L'algorithme met en valeur ce qu'un résolveur de contraintes est censé faire : résoudre un système d'inéquations en suivant un graphe de dépendance. Ici le graphe de dépendance est directement lié à la notion de *position*. On notera que l'algorithme d'Ayers n'est applicable que sur les programmes préalablement traduit en CPS (*Continuation Passing Style*). Dans ce contexte, les valeurs des nœuds d'application internes dans le programme sources sont transférées au paramètre formel d'une continuation générée par la transformation. Ainsi, pour les programmes traduits en CPS, l'analyse n'a pas à calculer les valeurs associées aux nœuds d'application, le troisième cas devient obsolète et donc l'algorithme d'Ayers n'a pas à construire la table d'association *Redexes*.

Pour l'analyse de complexité, commençons par les lignes *3* et *4* qui sont exécutées $\sum_{e \in \mathcal{E}} \|\mathcal{S}(e)\|$ fois. Par suite, la complexité de la ligne *5* est le nombre d'appels effectués à la ligne *4*, ce qui représente exactement $\sum_{e \in \mathcal{E}} \|\mathcal{S}(e)\| \times \|Pos(e)\|$ fois. Sachant que ce terme est en $\mathcal{O}(N^2)$ (voir la section 2.4), toutes les lignes définissant la procédure *UpdatePos* ne peuvent excéder cette complexité. Puisque chacune de ces lignes se calcule en temps linéaire, le nombre total d'appel à la procédure $UpdateValue$ est, au pire des cas, en $\mathcal{O}(N^3)$. Ceci nous donne la complexité générale de l'algorithme.

4. Les programmes *assassins*

Les programmes décrits dans cette section et leurs degrés de complexité sont résumés dans la figure 5. Pour l'algorithme naïf, nous avons testé trois manières différentes de collecter les nœuds d'application :

fac veut dire qu'un nœud d'application (c) apparaît après ceux se trouvant dans sa partie fonctionnelle (*f*) et dans son argument (a), *cfa* n'est pas vraiment intuitif puisqu'un nœud d'application sera visité avant ses sous-expressions, et enfin, *cfac* est une combinaison des deux où les nœuds d'application sont visités deux fois par itérations, nous verrons que cette façon de collecter les nœuds d'application ne donne pas forcément le minimum des complexités fournis par les méthodes *fac* et *cfa*.

Afin de simplifier la description des programmes, voici quelques combinateurs usuels :

$$
\begin{array}{ll}
I & \equiv \lambda x.x \\
True & \equiv \lambda x.\lambda y.x \\
False & \equiv \lambda x.\lambda y.y \\
\Delta & \equiv \lambda x.(x\ x) \\
Y & \equiv \lambda m.(\Delta\ \lambda f.(m\ \lambda x.(f\ f\ x)))
\end{array}
$$

Y est le combinateur de point fixe pour une stratégie d'appel par valeur. Il est utilisé pour la notation **letrec** faisant partie des raccoucis syntaxiques que voici :

$$
\begin{array}{ll}
\lambda x_1 x_2 \ldots x_n.B & \equiv \lambda x_1.\lambda x_2.\ldots\lambda x_n.B \\
(FA_1\ A_2 \ldots A_n) & \equiv (\ldots((F\ A_1)\ A_2)\ldots A_n) \\
A;B & \equiv (False\ A\ B) \\
(\textbf{if}\ P\ A\ B) & \equiv (P\ A\ B) \\
(\textbf{let}\ x = V\ \textbf{in}\ B) & \equiv (\lambda x.B\ V) \\
(\textbf{letrec}\ f = V\ \textbf{in}\ B) & \equiv (\textbf{let}\ f = (Y\ (\lambda f.V))\ \textbf{in}\ B)
\end{array}
$$

Pour commencer, nous avons besoin de N valeurs, ce qui sous-entend N fonctions. Le plus simple pour obtenir ces fonctions est de les chaîner entre-elle : $\lambda x_N.\ldots\lambda x_1. < Body >$. Nous prendrons la fonction identité comme corps de la dernière fonction, soit : $V_N = \lambda x_N \ldots x_1.I$. On peut dire que chaque fonction *cache* la suivante, de la même manière qu'un pointeur *cache* sa valeur. Le seul moyen de découvrir cette valeur cachée est d'appliquer la fonction, ce qui peut être fait, d'une manière très courte, en en utilisant le combinateur Δ. Ainsi, pour retrouver le corps de la dernière fonction (i.e. I) nous devons produire :

$$\text{Base}_N \equiv (\Delta_N \ldots (\Delta_1\ V_N)\ldots)$$

Cette expression relativement simple, correspondant au parcours d'une liste constante de N éléments, révèle une complexité de deux ordre en dessous de celui des pires cas pour l'ensemble des algorithmes, excepté pour celui influencé par l'évaluateur pour lequel l'ordre est seulement de 2. Cette expresssion ne produit pas un graphe dense, un tel graphe peut être obtenu en rassemblant tous les Δ_i dans une boucle :

$$\text{BaseRec}_N \equiv (\textbf{letrec}\ loop = \lambda v.(\textbf{if}\ ?\ v\ (loop\ (\Delta\ v)))\ \textbf{in}\ (loop\ V_N))$$

Où ? est une expression produisant les deux valeurs booléennes, par exemple : $? \equiv (\textbf{let}\ f = I\ \textbf{in}\ (f\ True);(f\ False))$. Ce programme BaseRec a un nombre constant de nœuds d'application et donc modifie peu les complexités exprimées par $Base_N$.

Le dernier effort pour produire les programmes *assassins* est d'insérer un appel à une fonction partagée F_N entre les appels au combinateur Δ de Base_N :

$$\text{Pattern}_N \equiv (\textbf{let}\ f = F_N\ \textbf{in}\ (\Delta_N\ (f\ \ldots(\Delta_1\ (f\ V_N))\ldots)))$$

La fonction F_N est censée utiliser son paramètre formel et retourner sa valeur pour l'application suivante à Δ. Puisque F_N n'apparaît qu'une seule fois dans Pattern_N, sa taille peut être en $\mathcal{O}(N)$. Mais commençons par une fonction de taille constante ($F_N = I$) qui, étrangement, révèle le pire cas de complexité pour l'algorithme orienté par la position.

Ensuite, une manière simple d'obtenir une fonction d'identité comportant $\mathcal{O}(N)$ sous-expressions et d'appeler l'opérateur I, N fois :

$$F_N = \text{I-chain}_N \equiv \lambda x.(I_N\ \ldots\ (I_1\ x)\ldots)$$

function	blind			eval	time	position
	fac	cfa	cfac			
Base	3	3	3	2	2	1
BaseRec	3	3	3	3	2	2
I	4	4	4	3	3	**3**
I-chain	4	**5**	4	4	3	**3**
Deep-let	**5**	**5**	4	4	**4**	**3**
Deep-let2	**5**	**5**	**5**	**5**	**4**	**3**
Rolling-left	**5**	4	4	**5**	**4**	**3**
Rolling-right	4	**5**	4	4	3	**3**

Figure 5: Order of complexity for the killer series

Cette série de programmes révèle le pire cas de complexité pour l'algorithme naïf lorsque les nœuds d'application sont pris dans l'ordre inverse. Ceci montre que l'ordre dans lequel sont pris les nœuds d'application peut produire des complexités différentes. Dans cette serie de programmes, les nœuds d'application sont toujours en position d'argument. Pour mettre les nœuds d'application en partie fonctionnelle il suffit de générer une chaîne de redex, ou, de manière équivalente, de produire une cascade de **let** :

$$\begin{aligned} F_N = \text{Deep-let}_N \quad & \equiv \lambda x.(\lambda x_N.(\lambda x_{N-1}.(\ldots((\lambda x_1.x_1)\ x_2)\ldots)\ x_N)\ x) \\ & \equiv \lambda x.(\mathbf{let}\ x_N = x\ \mathbf{in}\ (\mathbf{let}\ x_{N-1} = x_N\ \mathbf{in}\ \ldots(\mathbf{let}\ x_1 = x_2\ \mathbf{in}\ x_1)\ldots)) \end{aligned}$$

Ainsi, le redex le plus profond ($\mathbf{let}\ x_1 = x_2\ \mathbf{in}\ x_1$) agit sur un argument non encore analysé. Cette série peut aussi se définir par induction :

$$\begin{array}{lll} U_1 & \equiv \lambda x1.x1 & \\ U_i & \equiv \lambda x_i.(U_{i-1}\ x_i) & (i > 1) \\ \text{Deep-let}_N & \equiv \lambda x.(U_N\ x) & \end{array}$$

Elle fait apparaître les pires cas de complexité pour tous les algorithmes exceptés celui influencé par l'évaluateur et le naïf lorsque les nœuds d'application sont visités deux fois par itération (une fois avant et une fois après ses sous-expressions). Rapellons que l'algorithme orienté par l'évaluation est bloqué dans son parcours en profondeur lorsqu'une même fonction est appelée deux fois. Pour faire apparaître ce blocage, chaque fonction U_i doit être appelée au moins deux fois; une première fois avec un argument non significatif (disons I), puis avec l'argument fauteur de trouble. On définie la série Deep-let2$_N$ avec une nouvelle définition de U_i :

$$U_i \equiv \lambda x_i.(\mathbf{let}\ f = U_{i-1}\ \mathbf{in}\ (f\ I);(f\ x_i)) \qquad (i > 1)$$

Cette série achève tous les algorithmes en mettant en évidence leurs pires cas de complexité. Une série moins tortueuse achevant l'algorithme influencé par l'évaluateur se trouve dans la thèse d'Andrew Ayers [2], au lieu d'appeler deux fois les fonctions on utilise une récursion en prenant soin de faire tourner les arguments :

$$F_N = \text{rolling-left}_N \equiv \lambda x.(\mathbf{letrec}\ R = \lambda x_1 x_2 \ldots x_N.(\mathbf{if}\ ?\ x_1\ (R\ x_2 \ldots x_N\ x_1))\ \mathbf{in}\ (R\ I \ldots I\ x))$$

Cette série a le même effet que Deep-let2$_N$ excepté pour l'algorithme naïf lorsque les nœuds d'application sont analysés après leurs sous-expressions. Néanmoins, le sens, dans lequel les arguments tournent est important, on peut définir la dernière série par :

$$F_N = \text{rolling-right}_N \equiv \lambda x.(\mathbf{letrec}\ R = \lambda x_1 \ldots x_{N-1} x_N.(\mathbf{if}\ ?\ x_N\ (R\ x_N\ x_1 \ldots x_{N-1}))\ \mathbf{in}\ (R\ x\ I \ldots I))$$

Cette série retourne au même niveau que I-chain$_N$.

Bien sûr, lors de l'analyse de programme réel, on ne rencontre pas ce genre de programmes malsains. Néanmoins, les premières séries peuvent correspondre à des schémas de programme où N n'est généralement pas significatif.

On remarquera la chose suivante, qui restera la morale de cet article : plus rapide est *censé* être l'algorithme, plus simple est l'exemple qui exhibe le pire cas.

5. Autres analyses et travaux futurs

La 0CFA est une analyse très populaire mais dont l'origine reste un peu ambigüe. La première définition de la 0CFA, que l'on trouve dans le papier PLDI d'Olin Shivers [18] correspond strictement à la RSM restreint à un langage en forme CPS[4]. Par contre, dans sa thèse [19], la specification, toujours appelée 0CFA, est basée sur une sématique abstraite ayant besoin d'une notion d'environnement abstrait. Nous pensons que cette spécification n'est valide (*sound*) que pour les programmes CPS. De plus les preuves de correction (*soundness*) de la spécification deviennent plus hardues [5]. Enfin, la résolution directe de cette sémantique abstraite aboutit à un algorithme ne possédant pas la propriété de terminaison.

Une analyse qui pourrait être effectuée sur les λ-termes, consisterait à enlever l'itération de point fixe et de n'entrer dans le corps d'une fonction appelée que si une nouvelle valeur a été rajoutée à son paramètre formel ou à l'une de ses variables libres. Cette analyse est plus précise que la RSM et semble être la version non-CPS de la 0CFA définie dans la thèse de Shivers. Une des perspectives serait de prouver formellement la correction de cette analyse et d'en déterminer la complexité. Il semblerait que, bien que plus précise, cette analyse pourrait être plus rapide que les RSM usuellement implémentées.

6. Conclusion

Dans cet article nous avons présenté quatre algorithmes différents pour calculer la réduction statique minimale. À partir d'un algorithme *naïf* de vérification des règles de la RSM, nous avons décrit un algorithme influencé par l'évaluateur prenant les nœuds d'application en respectant, tant que possible, l'ordre qu'aurait pris un évaluateur. Cet algorithme est de type flot de contrôle puisqu'il essaye de suivre les chemins pris par le calcul dynamique. Ici, les valeurs sont prises comme un tout, en ne considérant que globalement l'ensemble dans lequel elles se trouvent, et donc en n'utilisant que les fonctions d'union et d'intersection sur les ensembles.

Puis, nous avons optimisé cet algorithme en introduisant un temps virtuel. Ceci nous a permis d'emprunter des raccourcis entre les itérations de point fixe : l'algorithme avec des valeurs triées évite, pour un nœud d'application particulier, d'analyser plusieurs fois la même fonction pour le même argument. Cet algorithme, comme le précédent, reste de type flot de contrôle mais s'intéresse plus spécifiquement aux valeurs lorsqu'elles sont appliquées ou transmises en argument. Un élément remarquable est que l'étude de la complexité de cet algorithme nécessite l'introduction de la notion de *position*. On remarquera aussi que la complexité de cet algorithme n'atteint $\mathcal{O}(N^3)$ que par la présence de la recherche de point fixe qui oblige à tout recommencer à chaque itération.

Ces considérations nous ont amené à penser que l'itération de point fixe était nécessaire car on abordait

[4] Par le fait que les nœuds d'application sont toujours en position terminale, seulement deux règles sont necessaires pour spécifier l'analyse.

[5] Seulement 200 lignes de Coq sont nécessaire pour la preuve de correction de la RSM, incluant la spécification du λ-calcul, de l'évaluateur, de l'analyse (les trois règles) et 45 lignes de preuves.

le problème en considérant le flot de contrôle (pour connaître les valeurs de telle expression, il faut connaître les valeurs de telle autre), en ramenant le problème sur le terrain du flot de données (cette valeur va passer par ici, puis par là), l'itération de point fixe disparaît pour faire apparaître une procédure récursive non terminale. Cet algorithme orienté par la position des valeurs atteint la complexité cubique attendue.

Comme il est précisé à la section 2, la RSM est censé calculer la solution *minimale* satisfaisant les règles décrites à la section 2.3. Si ce critère de minimalité n'est pas indispensable, un algorithme rapide, linéaire mais stupide serait : $\forall e \in \mathcal{E} : \mathcal{S}(e) \leftarrow \mathcal{L}$. On peut facilement vérifier que les trois règles fournies à la section 2.3 restent satisfaites. Entre cet algorithme sans intérêt et la RSM, beaucoup d'algorithmes restent à étudier.

7. Remerciements

La version finale de cet article a été influencée par les remarques des relecteurs anonymes. Nous remercions également Isabelle Attali et Andrew Ayers pour leurs commentaires.

Bibliographie

[1] J. Michael Ashley and R. Kent Dybvig. A practical and flexible flow analysis for higher-order languages. *ACM Transactions on Programming Languages and Systems*, 20(4):845–868, July 1998.

[2] A. Ayers. *Abstract Analysis and Optimization of Scheme*. PhD thesis, Massachusetts Institute of Technology, September 1993.

[3] G. Barthe and B. Serpette. Static reduction analysis for imperative object oriented languages. In M. Parigot and A. Voronkov, editors, *Proceedings of LPAR'00*, volume 1955 of *Lecture Notes on Computer Science*, pages 344–361. Springer-Verlag, 2000.

[4] Dominique Boucher and Marc Feeley. Abstract compilation: a new implementation paradigm for static analysis. In *In T. Gyimothy, editor, Proceedings of CC'96, the 6th International Conference on Compiler Construction, number 1060 in Lecture Notes in Computer Science*, Linkoping, Sweden, April 1996. Springer-Verlag.

[5] N. Heintze. *Set based program analysis*. CMU-CS-92-201, Carnegie Mellon University, Pittsburgh, Pa, 1992.

[6] N. Heintze and D. McAllester. Control-flow analysis for ML in linear time. In *1997 ACM Conference on Programming Language Design and Implementation*, 1997.

[7] Nevin Heintze. Control-flow analysis and type systems. In *Static Analysis Symposium*, pages 189–206, 1995.

[8] Nevin Heintze and David A. McAllester. Linear-time subtransitive control flow analysis. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 261–272, 1997.

[9] Nevin Heintze and David A. McAllester. On the complexity of set-based analysis. In *International Conference on Functional Programming*, pages 150–163, 1997.

[10] Nevin Heintze and David A. McAllester. On the cubic bottleneck in subtyping and flow analysis. In *Logic in Computer Science*, pages 342–351, 1997.

[11] Suresh Jagannathan and Stephen Weeks. A unified treatment of flow analysis in higher-order languages. In *Conference Record of POPL '95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 393–407, San Francisco, California, 1995.

[12] Christian Mossin. Exact flow analysis. In *Fourth International Static Analysis Symposium (SAS)*, pages 250–264, Paris, France, 1997. Springer-Verlag.

[13] F. Nielson and H. Seidl. Control-flow analysis in cubic time. In *Proc. ESOP'01*, number 2028 in Lecture Notes in Computer Science, pages 252–268. Springer-Verlag, 2001.

[14] Jens Palsberg. Closure analysis in constraint form. *ACM Transactions on Programming Languages and Systems*, 17(1):47–62, January 1995.

[15] Jens Palsberg and Patrick O'Keefe. A type system equivalent to flow analysis. *ACM Transactions on Programming Languages and Systems*, 17(4):576–599, July 1995.

[16] B. Serpette. Approximations d'évaluateurs fonctionnels. In *Proceedings of WSA (Workshop on Static Analysis)*, pages 79–90. Bigre, 1992.

[17] M. Serrano and M. Feeley. Sorage Use Analysis and its Applications. In *1fst Int'l Conf. on Functional Programming*, pages 50–61, Philadelphia, Penn, USA, May 1996.

[18] O. Shivers. Control flow analysis in scheme. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, Atlanta, Georgia, June 1988.

[19] O. Shivers. *Control-Flow Analysis of Higher-Order Languages or Taming Lambda.* CMU-CS-91-145, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, May 1991.

[20] O. Shivers. The Semantics of Scheme Control-Flow Analysis. In *Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, Yale University, New Haven, Conn., June 1991.

[21] Scott F. Smith and Tiejun Wang. Polyvariant flow analysis with constrained types. *Lecture Notes in Computer Science*, 1782:382–??, 2000.