TAS
Analyse d'article:
Compact Bit Encoding Schemes for Simply-Typed Lambda-Terms

Hugo Nakagawa Jordi Bertran de Balanda

Novembre 2016

Contents

1 Introduction

Nous présentons ici l'article *Compact Bit Encoding Schemes for Simply-Typed Lambda-Terms* [1]. Cet article présente deux approches de codage de λ-termes en chaînes binaires pour obtenir une représentation compacte d'un programme fonctionnel, adapté à la représentation de structures de données arborescentes.

Cet article fait suite aux travaux de Kobayashi et al. sur l'encodage de structures arborescentes en un programme fonctionnel (traduisible en λ-termes) représentant cette structure [2].

La compression d'ordre supérieur présente plusieurs avantages:

- En théorie, le ratio de compression peut être *très elevé*.
- Les données compressées peuvent être manipulées *sans décompression*.
- La compression d'ordre supérieur peut être utilisée pour émuler facilement d'autres schémas de compression.

Pour exploiter la puissance de compression de ces idées, il est nécessaire de savoir encoder de manière compacte des λ-termes. En particulier, pour tirer plein parti de l'émulation de schémas de compression avec l'encodage d'ordre supérieur, il est nécéssaire de minimiser le surcoût induit par la représentation de λ-termes pour pouvoir rivaliser avec les algorithmes 'classiques' de compression.

Des travaux d'étude de compression de λ-termes simplement typés ont déjà été réalisés par Tromp [3] ainsi que Vytiniotis et Kennedy [4, 5]. Takeda et al. prétendent que ces travaux sont inadaptés à la recherche de qualité de compression visée. Nous verrons par la suite que la qualité de compression de l'algorithme présenté est effectivement bien supérieure à celle des travaux déjà effectués.

Ainsi, le but de l'article est de trouver un couple de fonctions suivantes, avec $\mathcal{J}$ les triplets (Γ, M, τ):

$$(Enc, Dec) \in (\mathcal{J} \to 0,1*) \times (0,1* \rightharpoonup \mathcal{J})$$

2 Rappels

2.1 λ-calcul simplement typé

2.1.1 Typage

Le *lambda*-calcul simplement typé possède la syntaxe de typage suivante:

$\tau ::= \mathrm{o}$
$\quad | \quad \tau_1 \to \tau_2$

Ici, o est **l'unique** type admis par le système de types utilisé dans l'article. Nous nous basons sur ce système de types pour les encodages de λ-termes. Les encodages en bits décrits, en revanche, ne repose pas sur la présence d'un type unique, mais peut également être utilisé avec d'autres constructeurs de types, notamment des types de données algébriques.

2.1.2 Notation de De Bruijn

Les indices de De Bruijn décrivent la distance entre la variable typée et l'abstraction qui la lie, en nombre d'abstractions. Les constantes sont considérées comme des variables libres, également représentées avec des indices de De Bruijn. Lorsque leur nom est important, il est aisé d'en créer une liste, et d'encoder cette liste séparément pour la diffuser en tandem avec le programme auquel elles appartiennent.

Nous décrivons ainsi le langage que nous compressons de la manière suivante:

M ::= n
| | $\lambda : \tau.M$
| | $M_1 M_2$

Un λ-terme est:

- Un indice de De Bruijn
- Une abstraction sur une variable de type τ
- Une application de $ M_1 $ à $ M_2 $

3 Première approche: codage typé de λ-termes sans étiquette

3.1 Définition

Un codage de λ-termes sans étiquetage utilise les restrictions de types du λ-calcul simplement typé pour permettre d'omettre les étiquettes permettant de différencier variables, applications et abstractions dans une notation de De Bruijn. C'est la première étape de compression qui permet de réduire l'alphabet nécessaire pour encoder les termes. On applique un algorithme d'encodage de chaînes de caractères à ce résultat sans étiquettes pour obtenir l'encodage final.

3.2 Encodage et décodage de λ-termes en séquences

Comme expliqué en 2.2.2, on cherche à encoder chaque λ-terme en une séquence d'indices et de symboles d'abstraction de la forme λ_τ. Les informations de type propres aux abstractions sont suffisantes à elles seules pour retrouver la

sémantique du λ-terme à partir de la séquence d'indices. L'article présente deux exemples qui permettent de comprendre le principe intuitivement, avant de définir formellement les fonctions d'encodage et décodage, et prouver leur fondement.

3.2.1 Exemples

3.2.1.1 Application simple

Comme premier exemple, nous prenons le terme défini par:

- La séquence d'indices et la forme d'applications 0(011)(011)
- Le type de ce λ-terme o
- L'environnement de types $\Gamma : (o \to o \to o) \cdot o$

On encode ce terme en la séquence simple de ses indices de De Bruijn, et le terme devient "0011011".

Cette première approche décrite par l'article *garantit* que la forme originale du terme soit récupérable par une suite de déduction à partir de cette forme réduite.

1. Comme le premier index de la séquence est 0, on sait que la séquence se lira $0M_1...M_k$.
2. Comme 0 est de type $o \to o \to o$ et que le terme est de type o, on déduit que le terme est de la forme $0M_1M_2$, où M_1 et M_2 sont de type o.
3. Connaissant le type du premier élément, il nous reste 011011. Le premier index étant 0, on déduit que M_1 est de la forme $0M_{1,1}M_{1,2}$ où $M_{1,1}$ et $M_{1,2}$ sont de type o, et $M_{1,1}$ est encodé par un préfixe de la sous-séquence 11011
4. Le premier indice de cette séquence étant 1 de type o, on en déduit immédiatement que $M_{1,1}$ est la simple variable 1.
5. On sait alors que $M_{1,2}$ est encodé par un préfixe de la sous-séquence 1011.
6. De même qu'en 4, on obtient que $M_{1,2}$ représente également 1.
7. Par le même raisonnement qu'en 3-6, on obtient que M_2 représente la même chose que M_1
8. On obtient donc la décompression du terme en 0(011)(011).

3.2.1.2 Abstractions

Un deuxième exemple permet de voir la manière dont le type des abstractions est géré par l'encodage proposé par l'article, avec le terme:

- Représenté par $(\lambda_{o\to o\to o}.\lambda : o.(10)0)0$
- Typé $o \to o$
- Sous l'environnement de types $o \to o \to o$

La version encodée de ce terme est $\lambda_{(o\to o\to o)\to o\to o}.\lambda_{o\to o}1000$, obtenu comme précédemment en listant les λ et les variables définissant ce terme.

Afin de retrouver la forme originale du terme, on itère sur la tête de sa représentation compressée comme lors du premier exemple, en ajoutant une règle pour les abstractions.

1. Le premier élément du terme est $\lambda_{(o\to o\to o)}$: on en déduit que le terme est de la forme $(\lambda.M_0)M_1...M_k$, où k est potentiellement 0.
2. D'après l'information de type connue, $k = 1$, donc le terme est en fait de la forme $(\lambda.M_0)M_1$, où M_0 et M_1 ont des environnements de types différents puisqu'on introduit une variable supplémentaire dans le corps de M_0, là où on ne fait qu'appliquer le résultat de l'abstraction à M_1.
3. De la même manière, on retrouve que le terme $\lambda_{o\to o}1000$ a pour forme $(\lambda.M_{0,0})M_{0,1}...M_{0,k}$.
4. Avec l'information de type, on retrouve $k = 0$, ce qui nous donne M_0 de la forme $\lambda : o.M_{0,0}$, et $M_{0,0}$ de type o sous $o \cdot (o \to o \to o) \cdot (o \to o \to o)$

En faisant aboutir ce raisonnement, on obtient bien le λ-terme de départ.

Il est important de remarquer ici que le type annoté sur une abstraction *n'est pas* celui de la variable liée par l'abstraction, mais celui de l'abstraction résultante. En effet, annoter le type de la variable liée rend impossible la déduction de la forme des termes à partir des informations de type en présentant deux possibilités distinctes lors de la résolution de l'abstraction.

3.2.2 Formalisation

Après ces exemples, l'article formalise les fonctions d'encodage et décodage qui sont le but de l'article.

3.2.2.1 Encodage

La fonction d'encodage est définie comme une fonction récursive prenant un environnement de typage **TEnv** et une suite de λ-termes Λ, en application partielle en un nombre indéterminé de **Sym**, l'alphabet de codage employé.

$\texttt{EncS} \in \texttt{TEnv} \times \Lambda \rightharpoonup \texttt{Sym}^* :$
$\texttt{EncS}(\Gamma, n) = n$
$\texttt{EncS}(\Gamma, \lambda : \tau_0.M) = \lambda_{typeOf(\Gamma,\lambda:\tau_0.M))}.EncS(\tau_0 \cdot \Gamma, M)$
$\texttt{EncS}(\Gamma, M_1, M_2) = EncS(\Gamma, M_1) \cdot EncS(\Gamma, M_1)$

3.2.2.2 Décodage

La fonction de décodage est définie récursivement comme une fonction d'un triplet séquence de symboles, environnement et type du λ-terme en application partielle dans un couple λ-terme et séquence de symboles restante.

Plus formellement:

$\texttt{DecS} \in \texttt{Sym}^* \times \texttt{TEnv} \times \texttt{Ty} \rightharpoonup \Lambda \times \texttt{Sym}^*$
$\texttt{DecS}(\epsilon, \Gamma, \tau) = \texttt{fail}$
$\quad \texttt{if } \Gamma(n) \texttt{ is of the form } \tau_1 \to ... \to \tau_k \to \tau (k \geq 0)$
$\quad \texttt{then let } (M_1, s_1) = DecS(s_0, \Gamma, \tau_1) \texttt{ in}$
$\quad\quad ...$
$\quad \texttt{let } (M_k, s_k) = DecS(s_{k-1}, \Gamma, \tau_k) \texttt{ in}$
$\quad\quad (nM_1...M_k, s_k)$
$\quad \texttt{else fail}$
$\texttt{DecS}(\lambda_{\tau' \to \tau''}) =$
$\quad \texttt{if } \tau' \to \tau'' \texttt{ is of the form } \tau_1 \to ... \to \tau_k \to \tau (k \geq 0)$
$\quad \texttt{then let } (M_0, s_0) = DecS(s, \tau' \cdot \Gamma, \tau'') \texttt{ in}$
$\quad \texttt{let } (M_1, s_1) = DecS(s_0, \Gamma, \tau_1) \texttt{ in}$
$\quad\quad ...$
$\quad \texttt{let } (M_k, s_k) = DecS(s_{k-1}, \Gamma, \tau_k) \texttt{ in}$
$\quad\quad ((\lambda : \tau_1 . M_0) M_1 ... M_k, s_k)$
$\quad \texttt{else fail}$

On distingue 3 points d'erreur de l'algorithme, dénotant chacun un encodage représentant un λ-terme mal formé:

- lorsqu'on essaie de décoder le symbole de fin d'entrée, quels que soient l'environnement et le type du programme
- lorsqu'un indice de De Bruijn n'a aucune correspondance dans l'environnement de types
- lorsqu'on ne retrouve pas le type du programme dans le type d'une abstraction

3.2.2.3 Preuve

Comme pour tout système de codage, la propriété clé est l'équivalence entre un λ-terme et le résultat du décodage de l'encodage de celui-ci.

En particulier, pour prouver le bien-fondé des fonctions définies plus haut, l'article démontre la propriété suivante:

$$\forall s \in \mathbf{Sym}^*, DecS(EncS(\Gamma, M) \cdot s, \Gamma, \tau)$$

Autrement dit, on démontre que pour tout symbole de l'alphabet d'encodage, le résultat du décodage de l'encodage d'une séquence suivie de ce symbole, selon un environnement de types commun, est la même séquence suivie du même symbole. Cette preuve est faite par induction, avec le cas particulier où $s = \epsilon$, c'est-à dire où tous les λ-termes à encoder/décoder ont déjà été traités. Pour ceci, l'article examine les deux formes potentiels du λ-terme M:

- Une application, soit $M = nM_1...M_k$, avec k potentiellement nul
- Une abstraction, soit $(\lambda : \tau' . M_0) M_1 ... M_k$, avec k potentiellement nul

3.3 Encodage binaire

Par l'encodage présenté en 3.2, on peut maintenant convertir des λ-termes en triplets (s, Γ, τ). On désire compresser cette représentation de λ-termes avec un encodage binaire de chaînes de caractères, pour produire un résultat potentiellement transmissible à bas niveau.

Cette compression opère sur 4 éléments différents:

1. Une table de types, qui associe à chaque type un identifiant unique (concrètement, un entier)
2. L'environnement de types Γ et le type du λ-terme τ
3. La séquence s de symboles, c'est-à-dire les λ-termes ôtés de leurs étiquettes
4. Si les noms des variables libres est important, une table similaire à 1. associant les indices de De Bruijn desdites variables libres à leurs noms

L'objectif de ces opérations est d'obtenir une chaîne d'entiers pouvant être directement encodée par un algorithme standard.

3.3.1 Table de types

On définit à l'intérieur de la table de types le schéma d'encodage du type τ comme la séquence binaire $[\![\tau]\!]$:

$$[\![o]\!] = 0; [\![\tau_1 \to \tau_2]\!] = 1[\![\tau_1]\!][\![\tau_2]\!]$$

La table de types elle-même est la séquence $[\![\tau_0]\!]...[\![\tau_k]\!]$ représentant les types rencontrés dans s. C'est un code préfixe, pour lequel il n'y a donc pas besoin de séparateur. On assignera les identifiants i à chaque τ_i rencontrés dans l'ordre.

3.3.2 Environnement et type du λ-terme

De la même façon qu'en 3.3.1, on encode la séquence représentant l'environnement de types du programme en une suite $[\![\tau]\!][\![\tau_0]\!]...[\![\tau_n]\!]$ correspondant à $\Gamma = \tau_0...\tau_n$. Dans l'optique de la création d'un algorithme de compression, une liberté ici est de supposer la taille combinée de la table de types et de l'environnement de types petite devant celle de la représentation sans étiquettes du programme s, sous peine de devoir recompresser celles-ci avec un algorithme standard de compression de chaînes de caractères. En réalité, la compression présentée par l'article ne requiert pas cette étape supplémentaire lorsque le λ-terme encodé est grand, avec un nombre de types présents réduits.

3.3.3 Séquence de symboles

On opère deux phases de remplacement avant d'utiliser un algorithme usuel de compression de chaînes (dans le cas de l'article, un codage par intervalles adaptatif).

- On remplace chaque λ_τ par l'identifiant de τ, préparé en 3.3.1.
- On additionne m à chaque indice de variable n de s, où $m-1$ est l'indice de type le plus élevé obtenu par l'encodage de Γ en 3.3.2

Ceci produit une séquence de nombres naturels dont l'encodage se fait 'classiquement', les auteurs ayant choisi dans l'article un encodage par intervalles adaptatif.

Lors de cette étape, on peut réaliser une optimisation importante qui permet de réduire le nombre de bits nécéssaires pour décrire les variables de la séquence s. En effet, sachant le type de chaque variable à tout instant, on peut utiliser les informations de type pour exclure certains symboles ayant des types incompatibles avec la portion déjà traitée du λ-terme, et ainsi déterminer un *sous-ensemble* des indices associés à Γ de manière transparente.

Plus formellement: si les types qui peuvent se trouver à la position i sont $a_0...a_k$, avec $a_o < ... < a_k$, et l'indice qui s'y trouve est a_j, alors on peut renuméroter a_j en j avec la garantie de pouvoir retrouver ce type au décodage, l'encodage et le décodage donnant accès aux mêmes informations de type au même moment.

4 Deuxième approche : Codage basé sur des grammaires

4.1 Introduction

La première approche a transformé le λ-terme en une séquence de nombres. Elle ne permet pas d'obtenir une compression qui prend en compte la répartition des nombres dans la séquence. En exploitant le fait que certains index de de Bruijn ne peuvent pas apparaître dans certaines portions du code, on peut réduire localement la plage d'entiers à utiliser pour l'encodage, ce qui permet de réduire l'espace mémoire occupé par le résultat de la compression. Mais, via le codage sans étiquette, cela est impossible, à cause du fait que le compresseur backend est un simple compresseur de chaîne de caractères.

Pour éviter ce problème, les auteurs proposent d'utiliser à la place un compresseur prenant en paramètre une grammaire non contextuelle (Context-Free Grammar, CFG). Utiliser un CFG offre d'autres avantages : être sûr de ne compresser que des λ-termes simplement typés (sous réserve que la grammaire soit correctement formulée), et restreindre la syntaxe des termes considérés, pour encoder de

façon encore plus compacte. Un compresseur basé sur le schéma d'encodage de Cameron, lui-même une variation du codage arithmétique, a été créé.

4.2 Schéma d'encodage de Cameron pour le CFG

Si on utilise un compresseur de chaîne de caractères, des codes bit vont être alloués pour des combinaisons de termes qui n'apparaîtront pas. Par exemple, en considérant les expressions arithmétiques à notation infixe, un code bit pourrait être alloué pour "+aa". Définir un arbre syntaxique (parse tree) ou une suite de règles de dérivation permet de limiter les possibilités. Prenons comme exemple la grammaire suivante (cf. page 151) :

```
type exp = Plus of exp * fact | Factor of fact
and fact = Times of fact * var | Var of var
and var = A.
```

Supposons que l'on souhaite compresser l'expression arithmétique a + a * a. On peut l'exprimer ainsi : Plus(Factor(Var(A))),Times(Var(A),A)). Si on associe 0 à Plus et Times (qui sont les premières options des non-terminaux, exp et fact) et 1 à Factor et Var (qui sont les secondes options des non-terminaux), on encodera l'expression en 01101. Pour décoder 01101 et retrouver la forme originelle, il faut partir de "exp", puis appliquer une par une les règles de la grammaire. La règle à appliquer est déterminée en fonction de la cible (qui est le premier élément non terminal) et le numéro lu (0, 1, 1, 0 et 1 dans cet ordre). Voici un tableau qui explicite le décodage.

Expression	N. traité	Règle appliquée	Cible
exp	0	Plus	exp
exp + fact	1	Factor	exp
fact + fact	1	Var	fact
var + fact	0	Times	fact
var + fact * var	1	Var	fact
var + var * var			

A la fin, nous retrouvons donc bien l'expression a + a * a de départ. Par ailleurs, a * a + a aurait été encodé en 01011.

On peut sur le même principe encoder des λ-expressions, en réécrivant la grammaire comme une suite de règles de production. Le fait que le schéma de Cameron soit basé sur le codage arithmétique permet d'utiliser l'arithmétique de virgule fixe.

Il faut maintenant développer une grammaire représentant les λ-termes simplement typés, qui décrit les termes correctement typés.

4.3 Encodage basé sur la grammaire de λ-termes simplement typés

Les règles de typage du λ-calcul simplement typé peuvent être vus comme des règles de production de CFG. Cela permet de construire une suite finie de règles de production d'un terme donné (fini) à partir des règles de typage.

On peut réduire le nombre de règles de production nécessaires pour les variables en modifiant l'indexage de de Bruijn. Au lieu de seulement associer un numéro à chaque variable, on lui associe également un type, et on l'indexe par rapport à son type. Ainsi, lors du décodage, pour retrouver à quel λ une variable est reliée, il faut d'abord consulter son type, puis ensuite son index. Son index sera utilisé comme pour le codage sans étiquette, à ceci près qu'il ne faut prendre en compte que les λ ayant le même type que la variable.

Les règles de production pour les λ-termes simplement typés sont (cf. page 151) :

$$\text{(Grammaire 1)} \quad \begin{array}{lcl} M_\tau & \to & n_{\tau_1 \to \cdots \to \tau_k \to \tau} M_{\tau_1} \dots M_{\tau_k} \\ & | & (\lambda : \tau'.M_{\tau''}) M_{\tau_1} \dots M_{\tau_k} \end{array}$$

Ces règles sont des templates : il faut remplacer les τ par des types concrets avant d'obtenir les véritables règles. Cela correspond à la représentation sans étiquette de la partie précédente. Cependant, avec ce CFG, le codage arithmétique peut être utilisé pour prendre en compte la fréquence de chaque règle de production.

Cette grammaire peut permettre de former des termes invalides, car elle ne prend pas en compte les environnements de types. Ce qui devient possible en prenant en compte quelle règle de production peut être utilisée à quel moment, et ainsi éviter d'allouer des bits à des règles inutilisées. On peut soit éliminer une règle inutilisée lors de son encodage, en calculant sa probabilité d'utilisation, soit prendre en compte l'environnement de type explicitement lors de la conception de la grammaire. Cette dernière option n'a pas été envisagée, car le nombre de règles de production aurait trop augmenté (dans le cadre du schéma de Cameron) pour être intéressant.

La grammaire peut être optimisée en considérant la $\beta\eta$-équivalence.

4.4 Optimisations basées sur la $\beta\eta$-équivalence

On peut encoder sans faire de différence entre deux termes α-équivalents. Si on se trouve dans le cadre de la compression d'ordre supérieur, en plus de cela, on peut ne pas différencier deux termes β-équivalents, et/ou η-équivalents. L'η-conversion est l'ajout ou le retrait d'une abstraction sur une fonction. Par exemple, l'expression OCaml suivante :

```
let add x = x + 1;;
```

devient, après η-expansion :

```
let add = function x -> x + 1;;
```

On pourrait choisir de n'allouer des codes bit que aux formes normales, mais cela rallonge la taille de l'expression, ce qui n'est pas le but recherché. Ainsi, on peut établir un ensemble de règles qui transforment les λ-expressions en d'autres qui sont $\beta\eta$-équivalents, seulement lorsque le résultat a une longueur plus courte que l'expression originelle. On incorpore donc les règles suivantes (cf. page 152) :

- $(\lambda x.L)MN \to (\lambda x.LN)M$ (x non libre dans N)
- $(\lambda x.M)y \to [y/x]M$
- $(\lambda x.L)((\lambda y.M)N) \to (\lambda y.(\lambda x.L)M)N$ (y non libre dans L)

En prenant la première règle en compte, la grammaire devient (cf. page 152) :

(Grammaire 2)

$$\begin{array}{rcl} M_\tau & \to & n_{\tau_1 \to \cdots \to \tau_k \to \tau} M_{\tau_1} \dots M_{\tau_k} \\ & | & (\lambda : \tau'.M_\tau) M_{\tau'} \\ & | & \lambda : \tau'.M_{\tau''} \text{ (avec } \tau = \tau' \to \tau'') \end{array}$$

En intégrant les trois règles, la grammaire devient (cf. page 152) :

(Grammaire 3)

$$\begin{array}{rcl} M_\tau & \to & n_{\tau_1 \to \cdots \to \tau_k \to \tau} M_{\tau_1} \dots M_{\tau_k} \\ & | & (\lambda : \tau'.M_\tau) N_{\tau'} \\ & | & \lambda : \tau'.M_{\tau''} \text{ (avec } \tau = \tau' \to \tau'') \\ N_\tau & \to & n_{\tau_1 \to \cdots \to \tau_k \to \tau} M_{\tau_1} \dots M_{\tau_k} \text{ (avec } k > 0) \\ & | & \lambda : \tau'.M_{\tau''} \text{ (avec } \tau = \tau' \to \tau'') \end{array}$$

On souhaite aussi prendre en compte la β-équivalence suivante (sous réserve que L et M ne contiennent ni x ni y) (cf. page 152) :

```
(let x = L in let y = M in N) <-> (let y = M in let x
    ↪ = L in N)
```

Cela permettrait d'éviter d'allouer de l'espace pour la même variable deux fois. Mais un tel cas de figure est difficile à détecter via une grammaire, puisque les deux écritures sont syntaxiquement identiques. Une solution possible est d'imposer un ordre total sur les λ-termes, et de réordonner les "let", puis de prendre en compte cela lors du calcul des probabilités d'utilisation.

4.5 Représentation en bits de l'information grammaticale

La représentation en bits d'un λ-terme est obtenue par l'encodage arithmétique basée sur une grammaire, ainsi que l'information concernant la grammaire choisie. Un codeur backend adaptatif, prenant en compte une grammaire et les plages, a été créé. On peut ainsi prendre par exemple la Grammaire 2 ou 3 comme template. Cela nécessite de préparer des séquences de types, et des

maps indiquant quelles règles existent ou non, pour obtenir une suite de règles depuis le template. Les données sont ensuite représentées comme une séquence de bits, et est passée à un compresseur de chaîne de caractères.

5 Expérimentations

Les auteurs ont fait de nombreux tests examinant la qualité des résultats produits par leur approche. Utilisant des jeux de données divers (séquençage ADN, code source, abstracts de papiers scientifiques collectées massivement selon un mot clé, et fichiers XML tirés de travaux sur la compression de structures arborescentes), ils génèrent des λ-termes grâce à un compresseur d'ordre supérieur. Ils comparent leurs deux approches dans leurs différents états d'optimisation aux algorithmes précédemment publiés.

Ils montrent ainsi que leurs algorithmes semblent être, sur des entrées de taille raisonnable, souvent d'un ordre de grandeur plus efficaces en qualité qui est, sur leur jeu de tests, meilleure que celle des algorithmes existants. La seule ombre au tableau est leur performance décevante par rapport aux travaux existants [3, 4, 5] pour de très petits jeux de données (moins de 150 termes), expliquable l'overhead induit par la nécéssité de véhiculer soit les informations de type ou de grammaire relatives au programme.

La comparaison sur les mêmes jeux de données avec un algorithme 'classique' spécialisé dans la compression de structures arborescentes en grammaire SLCDF (Straight Line Context-Free), TreeRepair [6], permet de constater que les algorithmes basés sur la compression d'ordre supérieur sont compétitifs avec celui-ci dans des jeux de tests développés pour ce dernier.

Il aurait été intéressant d'avoir des informations comparatives de performances en temps des différents algorithmes présentés avec les algorithmes existants, que les auteurs n'ont visiblement pas jugé utile d'inclure. Nous aurions ainsi pu conclure sur l'efficacité temporelle de leurs algorithmes aussi bien que sur la qualité de leurs résultats.

6 Conclusion

Cet article de recherche présente deux techniques de compression de λ-expressions. La première, reposant sur la déduction de la forme des λ-termes à partir d'informations de types transmises avec le programme, est idéale pour de petits programmes. La seconde se base sur les grammaires non contextuelles, et est optimisée pour les expressions d'ordre supérieur, en prenant en compte la $\beta\eta$-équivalence.

Leurs résultats, malgré un overhead important qui rend leur utilisation peu indiquée pour des programmes courts, présentent d'excellents résultats sur des programmes de taille conséquente, même mis en compétition avec des algorithmes spécialisés dans une forme particulière de structures arborescentes.

Ces recherches permettent d'obtenir de meilleurs résultats en compression de λ-expressions d'ordre supérieur, comparés aux précédents. L'article ouvre également la voie à la possibilité d'appliquer ce style de codage à des données autres que des structures arborescentes en posant les fondations d'un schéma d'encodage performant et adaptable par différentes optimisations décrites dans l'article.

Références

[1] Kotaro Takeda, Naoki Kobayashi, Kazuya Yaguchi, and Ayumi Shinohara. Compact bit encoding schemes for simply-typed lambda-terms. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*, pages 146–157. ACM, 2016.

[2] Naoki Kobayashi, Kazutaka Matsuda, Ayumi Shinohara, and Kazuya Yaguchi. Functional programs as compressed data. *Higher-Order and Symbolic Computation*, 25(1):39–84, 2012.

[3] John Tromp. Binary lambda calculus and combinatory logic. *Cristian S. Calude*, page 237, 2007.

[4] Dimitrios Vytiniotis and Andrew J Kennedy. Functional pearl: every bit counts. In *ACM Sigplan Notices*, volume 45, pages 15–26. ACM, 2010.

[5] Andrew J Kennedy and Dimitrios Vytiniotis. Every bit counts: The binary representation of typed data and programs. *Journal of Functional Programming*, 22(4-5):529–573, 2012.

[6] Markus Lohrey, Sebastian Maneth, and Roy Mennicke. Tree structure compression with repair. In *2011 Data Compression Conference*, pages 353–362. IEEE, 2011.