Efficient Bracket Abstraction Using Iconic Representations for Combinators

Antoni Diller
School of Computer Science,
University of Birmingham,
Birmingham, B15 2TT, UK

A.R.Diller@cs.bham.ac.uk

September 2011

Abstract

Some fundamental properties of a new uni-variate bracket abstraction algorithm employing a string representation for combinators are established. In particular, if the input term has length n, where $n > 1$, the algorithm is called fewer than n times to produce the abstract. Furthermore, the space required to store the abstract, in the worst case, is of the order $O(n)$. This algorithm also has a number of features that make it worthy of further attention. When it is used to abstract a variables from an input term of length n, where $n > 1$, fewer than an new combinators are introduced into the abstract. However, the total size of the string representations of these combinators grows quadratically in the number of variables abstracted and the space required to store the abstract, in the worst case, is of the order $O(a^2n)$. Fortunately, a closely related single-sweep, multi-variate algorithm exists, using an array representation for combinators, which produces an abstract whose storage requirement, in the worst case, is of the order $O(an)$.

1 Introduction

Uni-variate bracket abstraction in combinatory logic is a syntactic operation which removes a single variable x from a term X, written $[x]X$, satisfying the property that $([x]X)P \rightarrow [P/x]X$, where the arrow represents reduction and $[P/x]X$ is the result of substituting P for all free occurrences of x in X. The abstract $[x]X$ plays a similar role in systems of combinatory logic to that played by $\lambda x.X$ in systems of the λ-calculus.

The first abstraction algorithm in combinatory logic was devised at the same time as the subject was created [12] and over the years a large number of abstraction algorithms have been proposed [8]. For several years after Turner [14] proposed an algorithm that could be used to implement a usable functional language [15] a great deal of research effort was directed to devising better and more efficient algorithms. However, other ways of implementing functional languages have become standard, because they produce faster implementations. Such

methods include those employing super-combinators [10]. However, research on abstraction algorithms continues.

A significant new development in the design of abstraction algorithms was heralded in the work of Stevens [13]. Before he presented his algorithm, all abstraction algorithms used non-iconic representations for combinators. In such a notation the connection between a combinator's name and its behaviour is given by an arbitrary stipulation. The names of the commonest combinators **S**, **K**, **I**, **B** and **C** are examples of non-iconic representations. The name **B**, for example, gives no indication of how the combinator **B** behaves; you have to be told how it behaves. Stevens [13] introduced the first iconic representations for combinators. In an iconic representation the behaviour of a combinator is read off from its representation. (Director strings [9] can be thought of as a precursor to iconic representations, but director strings are not names of combinators; the director-string calculus is a different formal system from combinatory logic.) Not long ago Broda and Damas devised an interesting and efficient algorithm [2] which, in effect, uses an iconic representation, but different from Stevens's. My work falls into this new area of research. However, my way of representing combinators differs from both that employed by Stevens and by Broda and Damas; it represents combinators as strings of the letters **y** and **n**.

2 Fixing Terminology

There are several systems of combinatory logic. The one used here is *weak* combinatory logic. On the whole, standard terminology is used [7]. Assume given an infinite sequence of symbols called *variables* and two constants, **K** and **S**, called *basic combinators*. The letters v, w, x, y and z, sometimes decorated with subscripts, are used for variables. An *atom* is a variable or a constant. A *term* is defined thus: (a) Every variable is a term; (b) Every constant is a term; (c) If P and Q are terms, so is (PQ). The letters E, F, G, H, K, P, Q, R, S, T, X and Y, sometimes decorated with subscripts and superscripts, are used for terms. A term of the form (PQ) is an *application*, but the outermost pair of parentheses is usually omitted. Normally, no space is left between the terms of an application, but sometimes one will be inserted for clarity and readability. Application associates to the left, so $PQRST$ is the same as $(((PQ)R)S)T$. The symbol $\equiv$ represents *syntactic identity*: $P \equiv Q$ means that P and Q are exactly the same term. Because combinatory logic contains no variable-binding operators every variable in a term is *free*: $FV(P)$ represents the set of free variables in P. The *length* of P, written $\#P$, is the number of occurrences of atoms in P. A *subterm* is defined thus: (a) P is a subterm of P; (b) P is a subterm of QR if P is a subterm of Q or P is a subterm of R. Every term P can be uniquely expressed in the form $P_1P_2 \ldots P_m$, where P_1 is an atom and $m \geq 1$. The P_i are known as the *primal components* of P [1, p. 223]. The non-standard notion of a *subprimal component* is defined thus: (a) P is a subprimal component of P; (b) P is a subprimal component of Q if P is a subprimal component of one of the *primal* components of Q. For example, the subprimal components of $vw(x(yz))$ are: $vw(x(yz))$, v, w, $x(yz)$, x, yz, y and z.

The non-standard notation $rp(P)$ denotes the number of distinct non-atomic subprimal components of P other than P itself. Thus, $rp\big(vw(x(yz))\big) = 2$. There is a simpler way to work out what $rp(P)$ is. If P is represented using the fewest possible parentheses, then $rp(P)$

is equal to the number of right parentheses in P. (The letters 'rp' stand for 'right parentheses'.) Let $P \equiv P_1P_2 \ldots P_m$, where P_1 is an atom. Then

$$rp(P) = \sum_{i=1}^{m} \textbf{if } \#P_i = 1 \textbf{ then } 0 \textbf{ else } 1 + rp(P_i).$$

Putting a conditional inside a summation may be unusual, but its meaning is straightforward and explained fully elsewhere [6, p. 312]. If $\#P \geq 2$, then the minimum value of $rp(P)$ is 0 and its maximum value is $\#P - 2$. This maximum value occurs when P looks like $v(w(x(yz)))$, say.

The non-standard notation $rpx(x, P)$ is used to denote the number of distinct non-atomic subprimal components of P, other than P itself, that contain at least one occurrence of x. For example, $rpx\big(x, vw(x(yz))\big) = 1$. If P is represented using the fewest possible parentheses, then $rpx(x, P)$ is equal to half the number of parentheses that enclose subprimal components containing the variable x. (A parenthesis is counted only once even if it encloses more that one occurrence of the variable x.) Let $P \equiv P_1P_2 \ldots P_m$, where P_1 is an atom. Then

$$rpx(x, P) = \sum_{i=1}^{m} \textbf{if } P_i \not\equiv x \textbf{ and } x \in \mathit{FV}(P_i) \textbf{ then } 1 + rpx(x, P_i) \textbf{ else } 0.$$

If $\#P \geq 2$, then the minimum value of $rpx(x, P)$ is 0 and its maximum value is $\#P - 2$. The maximum value occurs when P looks like $x(y(z(xx)))$, say. It should also be noted that $rpx(x, P) \leq rp(P)$, for all variables x and terms P. (Both $rp(P)$ and $rpx(x, P)$ are useful in proving results about algorithms using string combinators because of the way in which those algorithms work.)

A term of the form $\mathbf{K}PQ$ or $\mathbf{S}PQR$ is a *redex*. *Contracting* an instance of a redex in a term S means replacing one occurrence of $\mathbf{K}PQ$ by P or one occurrence of $\mathbf{S}PQR$ by $PR(QR)$. Let the result be T. Then we say that S *contracts* to T, written $S \to_1 T$, and that T is the *contractum*. The *arity* of a basic combinator is the least number of terms it has to be followed by for it to be possible to contract it. Thus, the arity of $\mathbf{K}$ is 2 and the arity of $\mathbf{S}$ is 3. S is said to *reduce* to T, written $S \to T$, iff T results from S by carrying out a finite (possibly zero) number of contractions. Combinators $\mathbf{I}$, $\mathbf{B}$, $\mathbf{B'}$, $\mathbf{C}$, $\mathbf{C'}$ and $\mathbf{S'}$ can be defined in terms of $\mathbf{K}$ and $\mathbf{S}$: $\mathbf{I} \triangleq \mathbf{SKK}$, $\mathbf{B} \triangleq \mathbf{S(KS)K}$, $\mathbf{B'} \triangleq \mathbf{BB}$, $\mathbf{C} \triangleq \mathbf{S(BBS)(KK)}$, $\mathbf{C'} \triangleq \mathbf{B(BC)B}$ and $\mathbf{S'} \triangleq \mathbf{B(B(BS)S)K}$. We then have: $\mathbf{I}P \to P$, $\mathbf{B}PQR \to P(QR)$, $\mathbf{B'}PQRS \to PQ(RS)$, $\mathbf{C}PQR \to PRQ$, $\mathbf{C'}PQRS \to P(QS)R$ and $\mathbf{S'}PQRS \to P(QS)(RS)$.

Substituting the term P for every free occurrence of x in X, written $[P/x]X$, is defined in the following way: (a) $[P/x]x \equiv P$; (b) $[P/x]Y \equiv Y$, if Y is an atom distinct from x; (c) $[P/x]QR \equiv ([P/x]Q)([P/x]R)$. *Uni-variate bracket abstraction* is a syntactic operation which removes a variable x from a term X, written $[x]X$, satisfying the property that $([x]X)P \to [P/x]X$. If $[x]X = Q$, then X is the *input term* and Q the *abstract*. *Multi-variate bracket abstraction*, written $[x_1, x_2, \ldots, x_a]X$, removes several variables from a term X. There are two types of multi-variate abstraction: in the *multi-sweep* variety we have $[x_1, x_2, \ldots, x_a]X \triangleq [x_1]([x_2](\ldots([x_a]X)\ldots))$, whereas in the *single-sweep* variety the a variables are abstracted simultaneously in a single process. In this paper, unless explicitly stated otherwise, bracket abstraction shall mean uni-variate abstraction.

There are many abstraction algorithms. Turner's was the first to be used in a viable implementation of a functional language [15]. The following is a slightly improved version of Turner's algorithm which elsewhere [5, p. 98] I have called *algorithm (C)*. In this algorithm x cannot occur in either E or F, but must occur in both X and Y. Furthermore, the clauses of the algorithm have to be applied in the order given:

$$\begin{aligned}
[x]E &= \mathsf{K}E,\\
[x]x &= \mathsf{I},\\
[x]Ex &= E,\\
[x]EFX &= \mathsf{B}'EF([x]X),\\
[x]EXF &= \mathsf{C}'E([x]X)F,\\
[x]EXY &= \mathsf{S}'E([x]X)([x]Y),\\
[x]EX &= \mathsf{B}E([x]X),\\
[x]XE &= \mathsf{C}([x]X)E,\\
[x]XY &= \mathsf{S}([x]X)([x]Y).
\end{aligned}$$

(Bunder [3, pp. 659–660] explains how this algorithm, which he calls (abcd′e′f′def), differs from that devised by Turner [14]. The differences are not important here.)

3 Contraction

The algorithm presented in the next section employs a non-standard notation for combinators. This represents them as strings of the letters **y** and **n**, called **yn**-*strings*. These are examples of what Stevens [13] calls *iconic representations*. (His approach also uses a string notation for combinators, but the meaning of the two notations is completely different.) Thus, the letters **y** and **n** are known as *iconic letters*. The letter ϕ is used for an arbitrary **yn**-string. $size(\phi)$ is the number of occurrences of **y** and **n** in ϕ and ϕ_i, for $1 \leq i \leq size(\phi)$, is the ith letter in ϕ. String concatenation is represented by juxtaposition. As **yn**-strings represent combinators, they are considered to be constants and, thus, atoms. Thus, if ϕ is a **yn**-string, then $\#\phi = 1$. (This is acceptable from a mathematical point of view, and it simplifies many proofs, but from a computing perspective we need to take into account the amount of space required to store a **yn**-string. I will say more about this below.) Let ϕ be a **yn**-string. Then a ϕ-*redex* is any term of the form $\phi P_1 P_2 \dots P_{m+1}$, where $m = size(\phi)$. Thus, the arity of any **yn**-string ϕ is $1 + size(\phi)$. *Contracting* an instance of a ϕ-redex in a term S means replacing one occurrence of $\phi P_1 P_2 \dots P_{m+1}$ by $Q_1 Q_2 \dots Q_m$, where, for $1 \leq i \leq m$,

$$Q_i \equiv \begin{cases} P_i\ P_{m+1}, & \text{if } \phi_i = \mathbf{y};\\ P_i, & \text{if } \phi_i = \mathbf{n}. \end{cases}$$

As an example, consider the **yn**-string **ynyyn**:

$$\mathbf{ynyyn}\ P_1\ P_2\ P_3\ P_4\ P_5\ P_6 \to P_1\ P_6\ P_2\ (P_3\ P_6)\ (P_4\ P_6)\ P_5.$$

All the combinators introduced above, except **I**, can be represented using **yn**-strings: **K** is **n**, **S** is **yy**, **B** is **ny**, **B′** is **nny**, **C** is **yn**, **C′** is **nyn** and **S′** is **nyy**.

4 Translation

There is some superficial resemblance between **yn**-strings and the director strings of Kennaway and Sleep [9]. There are, however, several significant differences. One is that Kennaway and Sleep's director-string calculus is a formal system completely different from combinatory logic, whereas the approach taken in this paper is to work within combinatory logic, but to represent combinators iconically. However, just because we can devise a method for contracting **yn**-strings, that does not mean that they really do represent combinators. It is possible to devise apparently meaningful contractions for non-existent combinators. For example, it can be proved that there is no combinator **A** such that $\mathbf{A}X$ contracts to **S**, when X is not an atom, and to **K**, when X is an atom. If there were such a combinator, we would have that $\mathbf{A}(\mathbf{I}x) \to \mathbf{S}$ and also that $\mathbf{A}(\mathbf{I}x) \to \mathbf{A}x \to \mathbf{K}$. As **K** and **S** are distinct, there is no such combinator as **A**.

To show that **yn**-strings really are combinators it is necessary to show how every **yn**-string can be translated into a combination of the constants **K** and **S**. First define the constants **B**, **I** and **C** in terms of **K** and **S**. Then define the combinators $\mathbf{B}_i$, for $i \geq 1$, as follows:

$$\mathbf{B}_i \stackrel{\wedge}{=} \begin{cases} \mathbf{B}, & \text{if } i = 1, \\ \mathbf{B}\ \mathbf{B}_{i-1}\ \mathbf{B}, & \text{if } i > 1. \end{cases}$$

From this definition we have that $\mathbf{B}_i x y_1 y_2 \ldots y_i z \to x(y_1 y_2 \ldots y_i z)$. Let ϕ be a **yn**-string. Then the translation proceeds as follows:

$$\begin{aligned} \mathit{trans}(\mathbf{y}) &= \mathbf{B}\ \mathbf{I}, \\ \mathit{trans}(\mathbf{n}) &= \mathbf{K}, \\ \mathit{trans}(\phi\mathbf{y}) &= \mathbf{B}_i\ \mathbf{S}\ \mathit{trans}(\phi), \quad \text{if } \mathit{size}(\phi) \geq 1, \\ \mathit{trans}(\phi\mathbf{n}) &= \mathbf{B}_i\ \mathbf{C}\ \mathit{trans}(\phi), \quad \text{if } \mathit{size}(\phi) \geq 1, \end{aligned}$$

where $i = \mathit{size}(\phi)$. For example, $\mathit{trans}(\mathbf{ynyyn}) = \mathbf{B}_4\ \mathbf{C}\ (\mathbf{B}_3\ \mathbf{S}\ (\mathbf{B}_2\ \mathbf{S}\ (\mathbf{B}_1\ \mathbf{C}\ (\mathbf{B}\ \mathbf{I}))))$.

Theorem 1 *The translation function trans is correct in the sense that if*

$$\phi\ P_1\ P_2\ \ldots\ P_m\ P_{m+1} \to Q_1\ Q_2\ \ldots\ Q_m,$$

where $m = \mathit{size}(\phi)$ and the Q_i, for $1 \leq i \leq m$, are as given above, then

$$\mathit{trans}(\phi)\ P_1\ P_2\ \ldots\ P_m\ P_{m+1} \to Q_1\ Q_2\ \ldots\ Q_m.$$

Proof In order to prove that the translation is correct it is necessary to show that

$$\mathit{trans}(\phi)\ P_1\ P_2\ \ldots\ P_m\ P_{m+1} \to Q_1\ Q_2\ \ldots\ Q_m,$$

where $m = \mathit{size}(\phi)$ and, for $1 \leq i \leq m$,

$$Q_i \equiv \begin{cases} P_i\ P_{m+1}, & \text{if } \phi_i = \mathbf{y}, \\ P_i, & \text{if } \phi_i = \mathbf{n}. \end{cases}$$

The proof is by induction on the size of $\phi_1\phi_2\ldots\phi_m$. In the base case $m = 1$. There are two cases to consider. Either $\phi_1 = \mathbf{y}$ or $\phi_1 = \mathbf{n}$. When $\phi_1 = \mathbf{y}$, *trans*($\mathbf{y}$) $P_1\ P_2 = \mathbf{B}\ \mathbf{I}\ P_1\ P_2 \to \mathbf{I}\ (P_1\ P_2) \to P_1\ P_2$. When $\phi_1 = \mathbf{n}$, *trans*($\mathbf{n}$) $P_1\ P_2 = \mathbf{K}\ P_1\ P_2 \to P_1$. Both of these accord with the behaviour of $\mathbf{y}$ and $\mathbf{n}$ given above. Thus, the base case has been established.

To prove the inductive step we first assume that the result holds for $m - 1$. There are two cases to consider. In the first we look at $\phi_1\phi_2\ldots\phi_{m-1}\mathbf{y}$ and in the second we consider $\phi_1\phi_2\ldots\phi_{m-1}\mathbf{n}$. In the first case we have:

$$\begin{aligned}
&\textit{trans}(\phi_1\phi_2\ldots\phi_{m-1}\mathbf{y})\ P_1\ \ldots\ P_m\ P_{m+1}\\
&= \mathbf{B}_{m-1}\ \mathbf{S}\ \textit{trans}(\phi_1\phi_2\ldots\phi_{m-1})\ P_1\ \ldots\ P_m\ P_{m+1}\\
&\to \mathbf{S}\ (\textit{trans}(\phi_1\phi_2\ldots\phi_{m-1})\ P_1\ \ldots\ P_{m-1})\ P_m\ P_{m+1}\\
&\to \textit{trans}(\phi_1\phi_2\ldots\phi_{m-1})\ P_1\ \ldots\ P_{m-1}\ P_{m+1}\ (P_m\ P_{m+1})\\
&\to Q_1\ \ldots\ Q_{m-1}\ (P_m\ P_{m+1}),
\end{aligned}$$

where, for $1 \le i \le m - 1$,

$$Q_i \equiv \begin{cases} P_i\ P_{m+1}, & \text{if } \phi_i = \mathbf{y};\\ P_i, & \text{if } \phi_i = \mathbf{n}; \end{cases}$$

by the inductive hypothesis. In the second case we have:

$$\begin{aligned}
&\textit{trans}(\phi_1\phi_2\ldots\phi_{m-1}\mathbf{n})\ P_1\ \ldots\ P_m\ P_{m+1}\\
&= \mathbf{B}_{m-1}\ \mathbf{C}\ \textit{trans}(\phi_1\phi_2\ldots\phi_{m-1})\ P_1\ \ldots\ P_m\ P_{m+1}\\
&\to \mathbf{C}\ (\textit{trans}(\phi_1\phi_2\ldots\phi_{m-1})\ P_1\ \ldots\ P_{m-1})\ P_m\ P_{m+1}\\
&\to \textit{trans}(\phi_1\phi_2\ldots\phi_{m-1})\ P_1\ \ldots\ P_{m-1}\ P_{m+1}\ P_m\\
&\to Q_1\ \ldots\ Q_{m-1}\ P_m,
\end{aligned}$$

where, for $1 \le i \le m - 1$,

$$Q_i \equiv \begin{cases} P_i\ P_{m+1}, & \text{if } \phi_i = \mathbf{y};\\ P_i, & \text{if } \phi_i = \mathbf{n}; \end{cases}$$

by the inductive hypothesis. Combining these two cases yields a rule for the reduction of *trans*($\phi_1\phi_2\ldots\phi_m$) which accords with the reduction rule for $\phi_1\phi_2\ldots\phi_m$ given above. The result follows by induction. QED.

Some people have seen a similarity between **yn**-strings and Broda–Damas combinators [2, 4]. There is a superficial similarity of notation. However, the principles underlying the two approaches are radically different. Unfortunately, I do not have space to discuss Broda–Damas combinators at length. I will just make two points: (1) The collection of Broda–Damas combinators and the set of combinators that can be represented by **yn**-strings are not identical and neither is a subset of the other. (2) The way in which Broda and Damas represent combinators is more complicated than mine. The set of Broda–Damas indices *BD* is defined thus: (a) The empty word ϵ and the letters b and c are members of *BD*. (b) If $\alpha, \beta \in$ *BD*, then

In this algorithm P_1 must be an atom. The variable x can occur in any of the primal components P_i, for $1 \leq i \leq m$, but it does not have to occur in any of them.

$$[x]\ P_1\ P_2\ \ldots\ P_m = \phi_1\phi_2\ldots\phi_m\ Q_1\ Q_2\ \ldots\ Q_m,$$

where ϕ is a **yn**-string and, for $1 \leq i \leq m$,

$$\begin{array}{ll} \phi_i = \mathbf{y} \text{ and } Q_i \equiv \mathbf{I}, & \text{if } P_i \equiv x, \\ \phi_i = \mathbf{y} \text{ and } Q_i = [x]\ P_i, & \text{if } P_i \not\equiv x, \text{ but } x \in FV(P_i), \\ \phi_i = \mathbf{n} \text{ and } Q_i \equiv P_i, & \text{if } x \notin FV(P_i). \end{array}$$

Figure 1: Algorithm (L).

$c\cdot\alpha, b\cdot\alpha, (\alpha,\beta) \in BD$. They usually omit the dot, which represents concatenation. Examples of indices are: cbb, $bcbb$, $cbcbcbbc$, $(\epsilon, bcbb)$, $(bcbc, (cc, bcbc))$ and $((bcc, cb), (cbc, (bb, cc)))$. These indices are, in effect, iconic representations of combinators, but Broda and Damas actually use the notation Φ_α, where α is an index, to indicate a combinator. They also use **K** as a basic combinator. In their notation, **I** is Φ_ϵ, **S** is $\Phi_{(b,b)}$ and **B** is Φ_{bb}. Combinators $\mathbf{B'}$, **C**, $\mathbf{C'}$ and $\mathbf{S'}$ cannot be expressed as single Broda–Damas combinators, but have to be defined.

5 Abstraction

The new uni-variate abstraction algorithm presented in this section employs **yn**-strings to refer to combinators. The combinator **I** is also used. Algorithm (L) is shown in Fig. 1. The reasons for insisting that P_1 is an atom and for excluding the clause $[x]Ex = E$, when $x \notin FV(E)$, will be explained below. The clause $[x]x = \mathbf{I}$ is absent from (L) in order to simplify certain proofs. Its presence would not alter (L)'s operation much as it would only be used when the input term was the variable x. When the input term X is such that $\#X > 1$ and $x \in FV(X)$, then the treatment of the occurrences of x in X would not be affected by the presence of the clause $[x]x = \mathbf{I}$.

An example of the application of (L) should make its operation clear:

$$\begin{aligned} [x]\ x\ (y\ z)\ (z\ y\ x)\ (z\ (x\ y)) &= \mathbf{ynyy\ I}\ (y\ z)\ ([x]\ z\ y\ x)\ ([x]\ z\ (x\ y)) \\ &= \mathbf{ynyy\ I}\ (y\ z)\ (\mathbf{nny}\ z\ y\ \mathbf{I})\ (\mathbf{ny}\ z\ ([x]\ x\ y)) \\ &= \mathbf{ynyy\ I}\ (y\ z)\ (\mathbf{nny}\ z\ y\ \mathbf{I})\ (\mathbf{ny}\ z\ (\mathbf{yn\ I}\ y)). \end{aligned}$$

In the context of abstraction, the letter **y** means that the primal component to which it corresponds contains the abstraction variable. (The letter ϕ_i, for $1 \leq i \leq m$, is said to *correspond* to the primal component P_i.) The letter **n** means that the primal component to which it corresponds does not contain the abstraction variable.

Some of the basic properties of (L) are stated in the following theorem.

Theorem 2 *Let P and Q be terms. Then algorithm (L) has the following properties:*

(a) If algorithm (L) is applied to P, it will terminate.

(b) Algorithm (L) has the property that $([x]P)Q \rightarrow [Q/x]P$.

(c) Using algorithm (L) we have that $\#([x]P) = 1 + \#P + rpx(x, P)$.

(d) If $\#P \geq 2$, then $\#([x]P) \leq 2 \times (\#P) - 1$.

Proof The proofs of parts (a), (b) and (c) are all by complete induction on the value of $rpx(x, P)$. Part (d) is a straightforward corollary of part (c) and the properties of $rpx(x, P)$. The proofs are all relatively straightforward and so only that of part (c) will be given here.

Proof of part (c): The proof is by complete induction on the value of $rpx(x, P)$. In the base case $rpx(x, P) = 0$. Let $P \equiv P_1P_2 \ldots P_m$, where P_1 is an atom. For $1 \leq i \leq m$, either $P_i \equiv x$ or P_i is a term such that $x \notin FV(P_i)$. Using (L) we have that $[x]P = \phi Q_1Q_2 \ldots Q_m$, where ϕ and the Q_i, for $1 \leq i \leq m$, are as specified in Fig. 1. When $P_i \equiv x$, then $Q_i \equiv \mathsf{I}$. When P_i is a term such that $x \notin FV(P_i)$, then $Q_i \equiv P_i$. In both cases $\#Q_i = \#P_i$, for $1 \leq i \leq m$. Thus, $\#([x]P) = 1 + \#P$. Thus the base case has been established.

In the inductive step $rpx(x, P) > 0$. Let $P \equiv P_1P_2 \ldots P_m$, where P_1 is an atom. For $1 \leq i \leq m$, either $P_i \equiv x$ or P_i is a term such that $x \notin FV(P_i)$ or $P_i \not\equiv x$ and $x \in FV(P_i)$. Using (L) we have that $[x]P = \phi Q_1Q_2 \ldots Q_m$, where ϕ and the Q_i, for $1 \leq i \leq m$, are as specified in Fig. 1. When $P_i \equiv x$, then $Q_i \equiv \mathsf{I}$. When P_i is a term such that $x \notin FV(P_i)$, then $Q_i \equiv P_i$. In both these cases $\#Q_i = \#P_i$. When $P_i \not\equiv x$ and $x \in FV(P_i)$, then $Q_i = [x]P_i$ and $\#Q_i = 1 + \#P_i + rpx(x, P_i)$, by the inductive hypothesis. We thus have, where A_i abbreviates '$x \equiv P_i$ **or** $x \notin FV(P_i)$':

$$\begin{aligned}\#([x]P) &= 1 + \sum_{i=1}^{m} \textbf{if } A_i \textbf{ then } \#P_i \textbf{ else } 1 + \#P_i + rpx(x, P_i) \\ &= 1 + \#P + \sum_{i=1}^{m} \textbf{if } A_i \textbf{ then } 0 \textbf{ else } 1 + rpx(x, P_i) \\ &= 1 + \#P + rpx(x, P),\end{aligned}$$

by properties of $rpx(x, P)$. The result follows by complete induction. QED.

The length of $[x]P$, if $\#P \geq 2$, has its largest value when $rpx(x, P)$ has its maximum value. For example, let $Q \equiv x(y(z(xx)))$. Then $[x]Q = \mathsf{yy\,I}\ (\mathsf{ny}\ y\ (\mathsf{ny}\ z\ (\mathsf{yy\,I\,I})))$ and $\#([x]Q) = 9$.

Result (2c) enables us to calculate the number of times that (L) is called in order to produce an abstract. When (L) is used to produce an abstract, each variable in the input term, distinct from the abstraction variable, reappears in the abstract. Each constant in the input term also reappears in the abstract. Each occurrence of the abstraction variable in the input term is replaced by an occurrence of the combinator **I** in the abstract. Thus, by result (2c), the number of new **yn**-strings introduced into the abstract is $1 + rpx(x, P)$. As one new **yn**-string is introduced into the abstract each time (L) is called, it is called a total of $1 + rpx(x, P)$ times. If $\#P = n$ and $n > 1$, this means that (L) is called at most $n - 1$ times.

So far I have been assuming that the length of a **yn**-string is one. This is reasonable from a mathematical point of view, but if we are interested in the space complexity of (L), then this

assumption needs amending. Let $il(x, P)$ be the number of new iconic letters introduced by (L) when it is used to produce $[x]P$. Then we have the following result:

Theorem 3 *Using (L),* $il(x, P) \leq \#P + rpx(x, P)$.

Proof The proof is by complete induction on the length of P. In the base case $\#P = 1$, so $P \equiv x$ or $P \equiv Y$, where Y is an atom distinct from x. When $P \equiv x$, $[x]P = \mathbf{y}\,\mathbf{I}$ and so $il(x, P) = 1$. When $P \equiv Y$, where Y is an atom distinct from x, $[x]P = \mathbf{n}\,Y$ and so $il(x, P) = 1$. As $rpx(x, P) = 0$ in both case, the base case has been established.

In the inductive step $\#P > 1$. Let $P \equiv P_1P_2 \ldots P_m$ be a term, where P_1 is an atom. Also, let B_i abbreviate '$P_i \not\equiv x$ **and** $x \in FV(P_i)$'. Then

$$\begin{aligned} il(x, P) &= \sum_{i=1}^{m} \textbf{if } B_i \textbf{ then } 1 + il(x, P_i) \textbf{ else } 1 \\ &\leq \sum_{i=1}^{m} \textbf{if } B_i \textbf{ then } 1 + \#P_i + rpx(x, P_i) \textbf{ else } \#P_i, \end{aligned}$$

by using the inductive hypotheses $il(x, P_i) \leq \#P_i + rpx(x, P_i)$, for $1 \leq i \leq n$, in the **then**-clause and the fact that $1 \leq \#P_i$ in the **else**-clause,

$$\begin{aligned} &\leq \sum_{i=1}^{m} \#P_i + \sum_{i=1}^{m} \textbf{if } B_i \textbf{ then } 1 + rpx(x, P_i) \textbf{ else } 0 \\ &\leq \#P + rpx(x, P), \end{aligned}$$

using the properties of rpx. Thus, the inductive step has been established and the result follows by complete induction. QED.

The maximum value that $rpx(x, P)$ can take, when $\#P = n$ and $n > 1$, is $n - 2$. Thus, by Theorem 3, the maximum number of iconic letters that can be introduced by using (L) is $2(n - 1)$. Assuming that each occurrence of an iconic letter requires the same storage space as a variable or non-iconic combinator, we have that the maximum amount of space required to store the abstract is $3n - 2$. Thus, in the worst case, the space required to store the abstract produced by (L) is of the order $O(n)$.

I have assumed that each occurrence of an iconic letter requires the same storage space as a variable or non-iconic combinator because Broda and Damas make a similar assumption in working out the complexity of their algorithm. They take into account all occurrences of parentheses, commas and the letters b and c in their indices [2, p. 737]. With this assumption their algorithm also produces an abstract which, in the worst case, has a storage requirement of the order $O(n)$. At first sight, it may thus appear that there is little to choose between their algorithm and mine. However, digging deeper we see that only considering the space required to store the abstracts produced by different algorithms is not an entirely reliable guide to their efficiency. The Broda–Damas algorithm is a two-stage one. In the first stage they use Schönfinkel's algorithm to produce an abstract. (Schönfinkel's algorithm is obtained from algorithm (C) by removing the three clauses involving $\mathbf{B}'$, $\mathbf{C}'$ and $\mathbf{S}'$.) In the second stage of

their abstraction process, they transform this intermediate term into one using their distinctive indexed combinators. It is this final term whose storage requirement, in the worst case, is of the order $O(n)$. Algorithm (L), however, produces an abstract which requires a similar amount of storage space directly, without producing any intermediate code. Furthermore, **yn**-strings are simpler to handle than Broda–Damas indices. They are, after all, just bit-strings and thus capable of being easily stored and manipulated by a computer program. It is true that Bunder's two expedited versions of the Broda–Damas algorithm do not produce the long-winded intermediate code produced by the original Broda–Damas algorithm. However, when applied to an input term which is not an atom, algorithm (L) is called fewer than half as many times, in the worst case, to produce an abstract than the better of Bunder's two expedited algorithms, as I will now show. Bunder's better algorithm is called exactly $2p - 1$ times if the abstraction variable occurs p times in the input term [4, p. 1850]. Thus, its worst behaviour occurs if the abstraction variable occurs n times in the input term, where n is the length of the input term. It is then called $2n - 1$ times. Algorithm (L) behaves worst when $rpx(x, P)$ has its maximum value, where x is the abstraction variable and P is the input term. In this case, (L) is called at most $n - 1$ times, where $n = \#P$.

6 Multi-sweep, Multi-variate Abstraction

I now turn my attention to using algorithm (L) to carry out multi-sweep, multi-variate abstraction. Here is an example:

$$\begin{aligned}
[x]\ [y]\ [z]\ x\ y\ z\ (z\ x\ z) &= [x]\ [y]\ \mathbf{nnyy}\ x\ y\ \mathbf{I}\ ([z]\ z\ x\ z)\\
&= [x]\ [y]\ \mathbf{nnyy}\ x\ y\ \mathbf{I}\ (\mathbf{yny\ I}\ x\ \mathbf{I})\\
&= [x]\ \mathbf{nnynn\ nnyy}\ x\ \mathbf{I\ I}\ ([y]\ \mathbf{yny\ I}\ x\ \mathbf{I})\\
&= [x]\ \mathbf{nnynn\ nnyy}\ x\ \mathbf{I\ I}\ (\mathbf{nnnn\ yny\ I}\ x\ \mathbf{I})\\
&= \mathbf{nnynny\ nnynn\ nnyy\ I\ I\ I}\ ([x]\ \mathbf{nnnn\ yny\ I}\ x\ \mathbf{I})\\
&= \mathbf{nnynny\ nnynn\ nnyy\ I\ I\ I}\ (\mathbf{nnnyn\ nnnn\ yny\ I\ I\ I}).
\end{aligned}$$

Result (b) of Lemma 4 is needed in the proof of Theorem 5, which establishes a fundamental property about using (L) to perform several abstractions one after the other.

Lemma 4 *Using algorithm (L) we have that (a)* $rp(P) = rp([x]P)$ *and that (b)* $rpx(y, P) = rpx(y, [x]P)$.

Proof The proof of both these results is quite similar and both proceed by complete induction on the length of P. The proof of part (b) is slightly more difficult, so only that will be given here. In the base case $\#P = 1$. We consider three cases. Either P is x or y or R, where R is an atom distinct from both x and y. In each of these cases $rpx(y, P) = 0$. If $P \equiv x$, then $[x]P = \mathbf{y\ I}$. If $P \equiv y$, then $[x]P = \mathbf{n}\ y$. If $P \equiv R$, then $[x]P = \mathbf{n}\ R$. In each case $rpx(y, [x]P) = 0$. Thus, the base case has been established.

In the inductive step we need to show that the result holds for P on the assumption that it holds for all terms R such that $1 \leq \#R < \#P$. Let $P \equiv P_1\ P_2\ \ldots\ P_m$, where P_1 is an atom.

Then, unpacking the meaning of rpx, we need to show that for each i, such that $1 \leq i \leq m$, the conditional

$$\textbf{if } P_i \not\equiv y \textbf{ and } y \in \mathit{FV}(P_i) \textbf{ then } 1 + rpx(y, P_i) \textbf{ else } 0$$

has the same value as the conditional

$$\textbf{if } Q_i \not\equiv y \textbf{ and } y \in \mathit{FV}(Q_i) \textbf{ then } 1 + rpx(y, Q_i) \textbf{ else } 0,$$

where $[x]\ P = \phi\ Q_1\ Q_2\ \ldots\ Q_m$ and ϕ and the Q_i, for $1 \leq i \leq m$, are as specified by algorithm (L) in Fig. 1. The following three cases are mutually exclusive: (i) $P_i \not\equiv y$ and $y \in \mathit{FV}(P_i)$, (ii) $P_i \equiv y$ and (iii) $y \notin \mathit{FV}(P_i)$. The strategy of the proof is to show that in each of these three cases the above two conditionals have the same value.

(i) When $P_i \not\equiv y$ and $y \in \mathit{FV}(P_i)$, there are two sub-cases to consider, namely either $x \notin \mathit{FV}(P_i)$ or $x \in \mathit{FV}(P_i)$ and $P_i \not\equiv x$. In the first of these, $Q_i \equiv P_i$, by (L), and thus $rpx(y, P_i) = rpx(y, Q_i)$. In the second sub-case, $Q_i = [x]P_i$ and so $Q_i \not\equiv y$ and $y \in \mathit{FV}(Q_i)$. Thus, $rpx(y, P_i) = rpx(y, Q_i)$, by the inductive hypothesis. Thus, in both sub-cases the values of the above conditionals are the same.

(ii) When $P_i \equiv y$, then $Q_i \equiv P_i$ and the value of both conditionals is 0.

(iii) When $y \notin \mathit{FV}(P_i)$, there are three sub-cases to consider, namely either $P_i \equiv x$ or $x \notin \mathit{FV}(P_i)$ or $P_i \not\equiv x$ and $x \in \mathit{FV}(P_i)$. In the first sub-case, $Q_i \equiv \mathsf{I}$, thus $y \notin \mathit{FV}(Q_i)$ and the value of both the above conditionals is 0. In the second sub-case, $Q_i \equiv P_i$ and the value of both the above conditionals is 0. In the third sub-case, $Q_i = [x]P_i$, where $P_i \not\equiv x$. Abstraction does not introduce variables into the abstract, so $y \notin \mathit{FV}(Q_i)$ and the value of both the above conditionals is 0.

This establishes the inductive step and the result follows by complete induction. QED.

Note that algorithm (L) would not have the property that $rp(P) = rp([x]P)$ if P_1 in the specification of its operation given in Fig. 1 was not required to be an atom. Let $Q \equiv yzw(xx)$. Then Q provides a counter-example when P_1 is taken to be yzw. We have that $[x]Q = \mathsf{ny}\ (y\ z\ w)\ (\mathsf{yy\ I\ I})$, so $rp(Q) = 1$, but $rp([x]Q) = 2$. Note also that (L) would not have the property that $rpx(y, P) = rpx(y, [x]P)$, if it contained the clause $[x]Ex = E$, where $x \notin \mathit{FV}(E)$. A counter-example is provided by $z(yx)$. We have that $rpx\big(y, z(yx)\big) = 1$, but $rpx\big(y, [x]z(yx)\big) = 0$ as $[x]z(yx)$ would be $\mathsf{ny}\ y$.

Theorem 5 *If $\#P \geq 2$ and the x_i, for $1 \leq i \leq a$ are distinct variables, then*

$$\#([x_a]([x_{a-1}](\ldots([x_2]([x_1]P))\ldots))) = a + \#P + \sum_{i=1}^{a} rpx(x_i, P).$$

Proof The proof is by induction on a. It makes use of part (c) of Theorem 2 and part (b) of Lemma 4. The proof is straightforward and so I do not include it here. QED.

Each time (L) is called it introduces one new yn-string into the abstract. Therefore, the total number of new yn-strings in the abstract is the number of times (L) was called to produce that abstract. Thus, when used to abstract a different variables from a term P, of length n,

where $n > 1$, (L) is called $a + \sum_{i=1}^{a} rpx(x_i, P)$ times. That is to say, (L) is called at most $a(n-1)$ times.

Although fewer than an new **yn**-strings are introduced into the abstract produced by (L) when it is used to abstract a different variables from a term of length n, where $n > 1$, a lot of space is required to store these **yn**-strings. To calculate the space complexity of (L) we need first, using Theorem 3, to calculate an upper bound for the number of new iconic letters introduced when (L) is used to abstract a different variables from a term P of length n, where $n > 1$. Let $T_0 \equiv P$ and let $T_i = [x_i]T_{i-1}$, for $1 \leq i \leq a$. Then the total number of new iconic letters introduced is $\sum_{i=1}^{a} il(x_i, T_{i-1})$, where $il(x_i, T_{i-1}) \leq \#T_{i-1} + rpx(x_i, T_{i-1})$, for $1 \leq i \leq a$. To appreciate how an upper bound for this summation can be calculated, consider the first three terms:

$$il(x_1, T_0) \leq \#T_0 + rpx(x_1, T_0),$$
$$il(x_2, T_1) \leq \#T_1 + rpx(x_2, T_1) \leq 1 + \#T_0 + rpx(x_1, T_0) + rpx(x_2, T_1),$$

by Theorem (2c),

$$il(x_3, T_2) \leq \#T_2 + rpx(x_3, T_2) \leq 1 + \#T_1 + rpx(x_2, T_1) + rpx(x_3, T_2),$$

by Theorem (2c),

$$\leq 1 + 1 + \#T_0 + rpx(x_1, T_0) + rpx(x_2, T_1) + rpx(x_3, T_2),$$

by Theorem (2c). Using the fact that $rpx(x, P) \leq rp(P)$, for all variables x and terms P, we have that

$$\sum_{i=1}^{a} il(x_i, T_{i-1}) \leq \sum_{i=1}^{a} (i-1) + \#T_0 + i \times rp(P)$$
$$\leq \frac{(a-1)a}{2} + an + \frac{a(a+1)(n-2)}{2},$$

using the fact that $rp(P) \leq n - 2$, when $\#P = n$ and $n > 1$. Every constant in the input term reappears in the abstract, as does each variable distinct from every abstraction variable. Occurrences of variables, in the input term, identical to any of the abstraction variables are replaced by occurrences of **I** in the abstract. Thus, taking the storage requirements of these constants and variables into account and assuming each iconic letter takes up the same storage space as a variable or non-iconic combinator, the maximum amount of space required to store the abstract is $(a-1)a/2+an+a(a+1)(n-2)/2$. Thus, the worst-case space complexity of (L) when used to abstract a different variables from a term of length n, where $n > 1$, is $O(a^2 n)$. This is disappointing. Algorithm (L) only introduces a small number of combinators when it is used, but the **yn**-strings that are introduced get longer and longer with each abstraction. This is a pity given how good (L) is at performing a single abstraction. The cause of the quadratic growth in the number of iconic letters introduced is not difficult to find. It is due solely to increasing numbers of the iconic letter **n** that are needed at the start of the **yn**-strings that are introduced with each abstraction. If these could be eliminated, then the number of iconic letters introduced would only grow linearly. Fortunately, it is possible to do this, but only by using an array representation for combinators, as I will now explain.

In this algorithm the variables x_i and x_j, for $1 \leq i, j \leq a$, in the bracket prefix $[x_1, x_2, \ldots, x_a]$ are the same iff $i = j$. Furthermore, P_1 must be an atom.

$$[x_1, x_2, \ldots, x_a]\ P_1\ P_2\ \ldots\ P_m = \gamma\ Q_1\ Q_2\ \ldots\ Q_m,$$

where γ is an **yn**-array and, for $1 \leq i \leq a$ and $1 \leq j \leq m$,

$$\gamma_{i,j} = \begin{cases} \mathbf{y}, & \text{if } x_i \in \mathit{FV}(P_j), \\ \mathbf{n}, & \text{otherwise;} \end{cases}$$

and, for $1 \leq j \leq m$,

$$Q_j \equiv \begin{cases} \mathbf{I}, & \text{if } P_j \equiv x_i, \text{ for some } i \text{ such that } 1 \leq i \leq a, \\ P_j, & \text{if } x_i \notin \mathit{FV}(P_j), \text{ for any } i \text{ such that } 1 \leq i \leq a, \\ [x_{f_j(1)}, x_{f_j(2)}, \ldots, x_{f_j(q_j)}]\ P_j, & \text{otherwise;} \end{cases}$$

where $q_j = tv([x_1, \ldots, x_a], P_j)$ and, for $1 \leq k \leq q_j$, $f_j(k) = inx(k, [x_1, \ldots, x_a], P_j)$.

Figure 2: Algorithm (M).

7 Single-sweep, Multi-variate Abstraction

Algorithm (M) is shown in Fig. 2. The main properties of (M) are proved elsewhere [6], but some new results will be mentioned here.

(M) uses an array representation for combinators. A **yn**-array is a two-dimensional matrix in which each component is either **y** or **n**. Let $[\overrightarrow{x}] = [x_1, x_2, \ldots, x_a]$. Then the value of the function $tv([\overrightarrow{x}], P)$ is the total number of variables in the list $\overrightarrow{x}$ that actually occur in the term P. For example, $tv([x_1, x_2, x_3], x_1x_3) = 2$. The function $inx(i, [\overrightarrow{x}], P)$ returns the index of the ith variable in the list $\overrightarrow{x}$ that occurs in P. For example, $inx(1, [x_1, x_2, x_3], x_2x_3(x_1x_2)) = 2$. The element $\gamma_{i,j}$ of the **yn**-array γ tells us whether or not x_i occurs in P_j. A letter **y** says that it does and an **n** tells us that it does not. An example should make its operation clear:

$$\begin{aligned} [x, y, z]\ x\ y\ z\ (z\ x\ z) &= \left|\begin{array}{cccc} \mathbf{y} & \mathbf{n} & \mathbf{n} & \mathbf{y} \\ \mathbf{n} & \mathbf{y} & \mathbf{n} & \mathbf{n} \\ \mathbf{n} & \mathbf{n} & \mathbf{y} & \mathbf{y} \end{array}\right| \mathbf{III}\ ([x, z]\ z\ x\ z) \\ &= \left|\begin{array}{cccc} \mathbf{y} & \mathbf{n} & \mathbf{n} & \mathbf{y} \\ \mathbf{n} & \mathbf{y} & \mathbf{n} & \mathbf{n} \\ \mathbf{n} & \mathbf{n} & \mathbf{y} & \mathbf{y} \end{array}\right| \mathbf{III} \left(\left|\begin{array}{ccc} \mathbf{n} & \mathbf{y} & \mathbf{n} \\ \mathbf{y} & \mathbf{n} & \mathbf{y} \end{array}\right| \mathbf{III}\right). \end{aligned}$$

There is a relation between **yn**-strings and **yn**-arrays. Let $P \equiv P_1\ P_2\ \ldots\ P_m$. Then

$$\begin{aligned} [x_1]\ P &= \beta_{1,1}\beta_{1,2}\ldots\beta_{1,m}\ Q_1^1\ Q_2^1\ \ldots\ Q_m^1, \\ [x_2]\ P &= \beta_{2,1}\beta_{2,2}\ldots\beta_{2,m}\ Q_1^2\ Q_2^2\ \ldots\ Q_m^2, \\ &\ldots \\ [x_a]\ P &= \beta_{a,1}\beta_{a,2}\ldots\beta_{a,m}\ Q_1^a\ Q_2^a\ \ldots\ Q_m^a, \end{aligned}$$

where the **yn**-strings and the Q^i_j, for $1 \leq i \leq a$ and $1 \leq j \leq m$, are as specified by algorithm (L) as shown in Fig. 1. But we also have that

$$[x_1, x_2, \ldots, x_a]\ P = \gamma\ Q_1\ Q_2\ \ldots\ Q_m,$$

where the **yn**-array γ and the Q_j, for $1 \leq j \leq m$ are as specified by (M) as shown in Fig. 2. The connection between these is that $\gamma_{i,j} = \beta_{i,j}$, for $1 \leq i \leq a$ and $1 \leq j \leq m$. (The connection between the Q_j and the Q^i_j is more complicated and is not relevant here.) This connection, which is not mentioned in my earlier paper [6], is a straightforward consequence of the definitions of (L) and (M).

It is clear that the number of iconic letters introduced by (M), when it is used to abstract a different variables, is exactly the same as the number of iconic letters introduced in the a separate applications of (L) shown in the previous paragraph. Thus, when (M) is used to abstract a variables from a term of length n, the maximum space required to store the abstract is $(3n-2)a$. Thus, in the worst case, the space required to store the abstract is of the order $O(an)$. This result is not found in my earlier paper [6]. It is only by using the results proved in this paper that the space complexity of (M) can be accurately stated. In that earlier paper, the size of the abstract produced by (M) was shown to be, in the worst case, of the order $O(an^2)$. (The space complexity of (M) follows from Theorem 3 and the fact that the a abstractions are performed simultaneously. Thus, the length of the input term is the same for all of them. Thus, $a \times \big(\#P + rp(P)\big)$ is an upper bound on the number of iconic letters introduced.)

8 Conclusion

In this paper some fundamental properties of a new uni-variate bracket abstraction algorithm using an iconic representation for combinators have been established. When the length of the input term is n, where $n > 1$, the space required to store the abstract produced by (L) is, in the worst case, of the order $O(n)$. This compares favourably with other well-known algorithms and is the same as that of the Broda–Damas algorithm. In the worst case, the space required to store the abstract produced by Turner's algorithm is of the order $O(n^2)$ [10, p. 279], that produced by the director-string algorithm [9] and by a typical super-combinator algorithm [10, p. 279] are both of the order $O(n \ln n)$ and that produced by the algorithm of Broda and Damas is of the order $O(n)$ [2, p. 737]. The multiplication factor in the case of the Broda–Damas algorithm is 3 as it is in the case of (L).

(L) also produces an abstract efficiently. If the length of the input term is n, where $n > 1$, the algorithm is always called fewer than n times. This compares favourably with Turner's algorithm, which is called, in the worst case, in the order of $O(n)$ times [10, p. 279] and with a typical super-combinator algorithm, which is called, in the worst case, in the order of $O(n \ln n)$ times [10, p. 279]. Furthermore, in the worst case for both algorithms, (L) is called fewer than half as many times as the better of Bunder's two expedited algorithms. (Figures for the other algorithms mentioned are not available.) Therefore, when (L) is used to perform a single abstraction, it is, for the reasons given, slightly better than any of the other uni-variate algorithms mentioned.

When (L) is used to abstract a different variables from a term of length n, where $n > 1$, the situation is not as good. In this case, the space required to store the abstract, in the worst case,

is of the order $O(a^2n)$. This is disappointing, and surprising, because (L) is so good at performing single abstractions and because fewer than an iconic representations are introduced into the abstract when it is used. Thus, the cause of the inefficiency is not that (L) introduces lots of combinators, but rather the increasing length of the **yn**-strings that are introduced. This inefficiency can be remedied by introducing an array representation for combinators, as I have done elsewhere [6], but the resulting algorithm, called (M), is no longer a multi-sweep, multi-variate one. It is, rather, a single-sweep, multi-variate algorithm. There is, however, a close connection between the **yn**-arrays produced by (M), when it is used to abstract a different variables from an input term, and the a **yn**-strings produced by (L) when it is applied separately to the same input term. In the worst case, the space required to store the abstract (M) produces is of the order $O(an)$. The space complexity of (M) can only be stated so accurately by making use of the results proved in this paper about (L). Unfortunately, there are no single-sweep, multi-variate algorithms employing Broda–Damas indices, director strings, supercombinators or Turner's long-reach combinators $\mathbf{B}'$, $\mathbf{C}'$ and $\mathbf{S}'$ with which to compare the performance of (M). There are very few single-sweep, multi-variate algorithms in the literature. Since the resurgence of interest in bracket abstraction, due to the influence of computer science, only two (apart from mine), to the best of my knowledge, have been published, namely those of Abdali [1] and Piperno [11]. Concerning the complexity of his algorithm, Abdali simply states that the size of the abstract his algorithm produces is proportional to the size of the input term [1, p. 222]. Piperno, like me, distinguishes between the length of the abstract produced by his algorithm and the space required to store that abstract [11, p. 49], but he only estimates the length in his paper and not the storage requirement. By Piperno's algorithm, the *length* of the abstract $[\overrightarrow{x}]P$ is less than or equal to $\frac{5}{2}n - 2$, where $n = \#P$ [11, p. 49], whereas for algorithm (M), the *length* of the abstract $[\overrightarrow{x}]P$ is less than or equal to $2n-1$ [6, p. 316]. Thus, (M) is slightly better.

I readily admit that, although slightly better than the best of their rivals, algorithms (L) and (M) are not huge improvements over them. However, I believe they have several features that make them interesting and worthy of further consideration. There is, for example, a natural correspondence between **yn**-*strings* and the structure of a term of combinatory logic. Furthermore, the connection between the use of (L) to perform multi-sweep, multi-variate abstraction and the use of (M) to carry out single-sweep, multi-variate abstraction is unique in the world of bracket abstraction.

References

[1] S. K. Abdali. An abstraction algorithm for combinatory logic. *The Journal of Symbolic Logic*, 41:222–224, 1976.

[2] Sabine Broda and Luis Damas. Compact bracket abstraction in combinatory logic. *The Journal of Symbolic Logic*, 62(3):729–740, September 1997.

[3] M. W. Bunder. Some improvements to Turner's algorithm for bracket abstraction. *The Journal of Symbolic Logic*, 55:656–669, 1990.

[4] Martin Bunder. Expedited Broda–Damas bracket abstraction. *The Journal of Symbolic Logic*, 65(4):1850–1857, December 2000.

[5] Antoni Diller. *Compiling Functional Languages*. Wiley, Chichester, 1988.

[6] Antoni Diller. Efficient multi-variate abstraction using an array representation for combinators. *Information Processing Letters*, 84:311–317, 2002.

[7] J. Roger Hindley and Jonathan P. Seldin. *Introduction to Combinators and λ-calculus*. Cambridge University Press, Cambridge, 1986. London Mathematical Society Student Texts, vol. 1.

[8] M. S. Joy, V. J. Rayward-Smith, and F. W. Burton. Efficient combinator code. *Computer Languages*, 10:211–224, 1985.

[9] J. R. Kennaway and M. R. Sleep. Variable abstraction in $O(n \log n)$ space. *Information Processing Letters*, 24:343–349, 1987.

[10] S. L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice-Hall International series in computer science. Prentice-Hall International, London, 1987.

[11] A. Piperno. A compositive abstraction algorithm for combinatory logic. In H. Ehrig, R. Kowalski, G. Levi, and U. Montanari, editors, *TAPSOFT '87*, volume 250 of *Lecture Notes in Computer Science*, pages 39–51, Berlin, 1987. Springer-Verlag.

[12] M. Schönfinkel. Über die bausteine der mathematischen logik. *Mathematische Annalen*, 92:305–316, 1924.

[13] David Stevens. Variable substitution with iconic combinators. In Andrzej M. Borzyszkowski and Stefan Sokołowski, editors, *Mathematical Foundations of Computer Science*, volume 711 of *Lecture Notes in Computer Science*, pages 724–733, Berlin, 1993. Springer-Verlag.

[14] David A. Turner. Another algorithm for bracket abstraction. *The Journal of Symbolic Logic*, 44:267–270, 1979.

[15] David A. Turner. A new implementation technique for applicative languages. *Software—Practice and Experience*, 9:31–49, 1979.