Investigations into Iconic Representations of Combinators

Dr Antoni Diller
School of Computer Science
University of Birmingham
Birmingham
B15 2TT
England

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

Abstract

Various properties of two bracket abstraction algorithms, previously introduced, are discussed and some of them are proved in this paper. Algorithm (L) is uni-variate and uses yes-no representations, whereas algorithm (M) is multi-variate and uses array representations. It is shown that (a) both algorithms are structure-preserving, (b) there is a straightforward connection between $[x]\ P$ and $[x]\ Q$, produced by algorithm (L), if the primal components of Q, except the first, are a permutation of the primal components of P, except the first, (c) there is a simple connection between the two abstracts produced when algorithm (L) is used repeatedly on the same input term, but the abstractions are performed in a different order in each case, (d) there is a straightforward connection between the array representations produced by algorithm (M) and the yes-no representations produced when algorithm (L) is used to abstract the same variables individually from the same input term and (e) using algorithm (M) multi-variate abstraction can be "partitioned".

Keywords

Bracket abstraction, combinatory logic, functional programming.

1 Introduction

Bracket abstraction in weak combinatory logic is a syntactic operation which removes a variable x from a term X. It is usually represented as $[x]\ X$. My interest in abstraction algorithms has its origin in their use in the implementation of pure functional programming languages [1]. I am well aware that this way of implementing a functional language is out of fashion, but for quite a while I have believed that the full potential of combinator-based methods is still to be fully realised and that there are still many interesting things to be discovered about combinators and abstraction algorithms. This faith of mine was vindicated when Stevens [6] developed a family of novel abstraction algorithms. His approach involves referring to combinators by means of what he calls 'iconic representations'. Unlike the standard notation for combinators, which uses single-letter identifiers, iconic representations are multi-letter identifiers. They have a significant internal structure from which the reduction property of the combinator referred to can be read off. Inspired by Stevens's work I developed further new abstraction algorithms.

In this algorithm P_1 must be an atom.

$$[x]\ P_1\ P_2\ \dots\ P_m = \beta_1\beta_2\dots\beta_m\ Q_1\ Q_2\ \dots\ Q_m,$$

where β is a yes-no representation and, for $1 \le i \le m$,

$$\begin{array}{llll} \beta_i = \mathsf{y} & \text{and} & Q_i = \mathsf{I}, & \text{if } P_i = x; \\ \beta_i = \mathsf{y} & \text{and} & Q_i = [x]\ P_i, & \text{if } P_i \neq x, \text{ but } x \in FV(P_i); \\ \beta_i = \mathsf{n} & \text{and} & Q_i = P_i, & \text{if } x \notin FV(P_i). \end{array}$$

The reduction rule for the yes-no representation β introduced by the above algorithm is as follows:

$$\beta_1\beta_2\dots\beta_m\ P_1\ P_2\ \dots\ P_m\ P_{m+1} \to Q_1\ Q_2\ \dots\ Q_m,$$

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

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

Figure 1: Algorithm (L) and yes-no reduction.

Some of these made use of yes-no representations [3] and others array representations [2]. Because of similarities between all these new representations, the word ‘iconic’ is now used generically to refer to all of them.

Although the original motivation for developing iconic representations and their associated abstraction algorithms was that of improving the way in which pure functional languages were implemented, the present paper concentrates on a number of surprising properties possessed by yes-no and array representations and on an interesting connection between them. Some interesting properties of two algorithms, namely (L) and (M), that employ yes-no and array representations, respectively, are also proved. In particular, I show that (a) both algorithms are structure-preserving, (b) if P and Q are such that the primal components of Q, except the first, are a permutation of the primal components of P, except the first, then there is a straightforward connection between $[x]\ P$ and $[x]\ Q$, where both are obtained by using algorithm (L), (c) there is a straightforward connection between $[x_1]([x_2](\cdots([x_a]\ P)\cdots))$ and $[x_{\sigma(1)}]([x_{\sigma(2)}](\cdots([x_{\sigma(a)}]\ P)\cdots))$, where σ is a permutation of the numbers $\{1, 2, \dots, a\}$, (d) there is a straightforward connection between the array representations produced by algorithm (M) and the yes-no representations produced when algorithm (L) is used to abstract the same variables individually from the same input term and (e) using algorithm (M) multi-variate abstraction can be “partitioned”.

2 Fixing Terminology

Standard combinatory-logic terminology is used in this paper [4]. The letters E, K, P, Q, R, T and X, sometimes decorated with subscripts, superscripts or primes, are used for arbitrary terms. The letters x, y and z, sometimes decorated with subscripts or superscripts, are used as variables. Note that every term P can be uniquely expressed in the form $P_1\ P_2\ \dots\ P_m$, where P_1 is an atom and $m \ge 1$. The P_i are known as the *primal components* of P.

The notation used for combinators in this paper is unusual. In the case of algorithm (L), shown in Fig. 1, the combinators used are represented as strings of the letters **y** or **n** and in the case of algorithm (M), shown in Fig. 2, the combinators used are represented as arrays in which each element is either **y** or **n**. The *length* of a term P, denoted as $\#P$, is the number of occurrences of atoms that it contains. Note that if β is a yes-no representation, it is assumed that $\#\beta = 1$. Similarly, if γ is an array representation, it is assumed that $\#\gamma = 1$, but I discuss this assumption more fully elsewhere [2].

Let P be a term which, if $\#P > 1$, is represented as a binary tree whose internal nodes are application nodes. Furthermore, let us say that an application node *covers* a variable x if x occurs in the subterm whose main operator is the application corresponding to that node. Then $rpv(\{x_1, x_2, \ldots, x_a\}, P)$ is the number of application nodes in P whose right child is an application node which also covers at least one of the variables in the list $x_1, x_2, \ldots, x_a$. If P is represented in a linear form using the fewest possible parentheses, then $rpv(\{x_1, x_2, \ldots, x_a\}, P)$ is equal to half the number of parentheses that enclose subterms containing at least one of the variables in the list $x_1, x_2, \ldots, x_a$. For example, $rpv(\{x, y, z\}, x(yz)(wv)z) = 1$ and $rpv(\{x, y\}, x(y(wv))(w(xv))) = 3$. Let $P = P_1\ P_2\ \ldots\ P_m$, where P_1 is an atom and let $S = \{x_1, x_2, \ldots, x_a\}$. Then

$$rpv(S, P) = \sum_{j=1}^{m} \textbf{if } (\forall i \in 1..a)\ x_i \notin FV(P_j) \textbf{ or } (\exists i \in 1..a)\ x_i = P_j \\ \textbf{then } 0 \textbf{ else } 1 + rpv(S, P_j),$$

where '$\forall i \in 1..a$' means 'for all whole numbers i between 1 and a inclusive' and '$\exists i \in 1..a$' means 'for some whole number i between 1 and a inclusive'.

3 Algorithm (L)

The reduction rule for yes-no representations is given in Fig. 1. An example should make clear how it works:

$$\textsf{\textbf{ynyn}}\ P_1\ P_2\ P_3\ P_4\ P_5 \to P_1\ P_5\ P_2\ (P_3\ P_5)\ P_4.$$

Algorithm (L), which makes use of yes-no representation, is also shown in Fig. 1. An example should clarify its operation:

$$\begin{aligned}[x]\ x\ (y\ z)\ (y\ x\ z\ (x\ y))\ z &= \textsf{\textbf{ynyn}}\ \textsf{\textbf{I}}\ (y\ z)\ ([x]\ y\ x\ z\ (x\ y))\ z \\ &= \textsf{\textbf{ynyn}}\ \textsf{\textbf{I}}\ (y\ z)\ (\textsf{\textbf{nyn}}\ y\ \textsf{\textbf{I}}\ z\ ([x]\ x\ y))\ z \\ &= \textsf{\textbf{ynyn}}\ \textsf{\textbf{I}}\ (y\ z)\ (\textsf{\textbf{nyn}}\ y\ \textsf{\textbf{I}}\ z\ (\textsf{\textbf{yn}}\ \textsf{\textbf{I}}\ y))\ z.\end{aligned}$$

The basic properties of algorithm (L), such as its complexity, are proved elsewhere [3]. It is also shown there how yes-no representations can be translated into the standard notation for combinators. Many people, on first encountering yes-no representations, think that they are similar to director strings [5], but director strings are not combinators and to manipulate them a novel formal system called 'the director-string calculus' has to be developed. By contrast, yes-no representations are just another notation for combinators and so the whole machinery of combinatory logic can be used to manipulate them.

4 Algorithm (M)

The reduction rule for array representations is given in Fig. 2. If β is a yes-no representation or, equivalently, an $a \times 1$ or $1 \times m$ array representation, then $yc(\beta)$ is the number of occurrences of the letter **y** in β and $posy(i, \beta)$ is the position of the ith occurrence of **y** in β, where $1 \le i \le yc(\beta)$. If $i > yc(\beta)$, then $posy(i, \beta)$ is not defined. For example, $yc(\textsf{\textbf{ynnynyy}}) = 4$, 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 $i = j$ and 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 array representation and, for $1 \leq i \leq a$ and $1 \leq j \leq m$,

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

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

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

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

The reduction rule for the array representation γ introduced by the above algorithm is as follows:

$$\gamma\ P_1\ P_2\ \ldots\ P_m\ P_{m+1}\ \ldots\ P_{m+a} \rightarrow Q_1\ Q_2\ \ldots\ Q_m,$$

where, for $1 \leq j \leq m$, $Q_j = P_j\ P_{g_j(1)}\ P_{g_j(2)}\ \ldots\ P_{g_j(s_j)}$, where $s_j = \mathit{yc}(\gamma_{1,j}\gamma_{2,j}\cdots\gamma_{a,j})$ and, for $1 \leq k \leq s_j$, $g_j(k) = a + \mathit{posy}(k, \gamma_{1,j}\gamma_{2,j}\cdots\gamma_{a,j})$.

Figure 2: Algorithm (M) and array reduction.

$\mathit{posy}(1, \mathsf{ynnynyy}) = 1$ and $\mathit{posy}(2, \mathsf{ynnynyy}) = 4$. An example should clarify the way in which array representations are reduced:

$$\left|\begin{array}{cccc} \mathsf{n} & \mathsf{n} & \mathsf{y} & \mathsf{n} \\ \mathsf{y} & \mathsf{n} & \mathsf{y} & \mathsf{n} \\ \mathsf{n} & \mathsf{n} & \mathsf{y} & \mathsf{y} \end{array}\right|\ P_1\ P_2\ P_3\ P_4\ P_5\ P_6\ P_7 = P_1\ P_6\ P_2\ (P_3\ P_5\ P_6\ P_7)\ (P_4\ P_7).$$

The order of an $a \times m$ array representation γ is $a + m$ and the jth column of the array γ tells us which of the arguments P_{m+1}, P_{m+2}, $\ldots$, P_{m+a} appear with P_j in the contractum of the array representation: P_{m+i} only occurs if $\gamma_{i,j}$ is y. For example, the 4th column of the array representation used in the example is nny. This means that $P_4\ P_7$ occurs in the contractum.

An abstraction algorithm that uses array representations is shown in Fig. 2. The main properties of algorithm (M), such as its complexity, are proved elsewhere [2]. It is also shown there how array representations can be translated into the standard notation for combinators.

Whenever multi-variate abstraction $[x_1, x_2, \ldots, x_a]\ P$ is mentioned in this paper, it is assumed that all the variables in the bracket prefix are distinct. Note that a different algorithm would result if it was not a requirement for P_1 to be an atom. The function $\mathit{tv}([\vec{x}], P)$ returns the total number of variables in the list $\vec{x}$ that actually occur in the term P. For example, $\mathit{tv}([x_1, x_2, x_3], x_1x_3) = 2$. The function $\mathit{inx}(i, [\vec{x}], P)$ returns the index of the ith variable in the list $\vec{x}$ that occurs in P. For example, $\mathit{inx}(1, [x_1, x_2, x_3], x_2x_3(x_1x_2)) = 2$. The element $\gamma_{i,j}$ of the array representation 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 of the use of algorithm (M) should make its operation clear:

$$\begin{aligned}[x,y,z]\ x(yx)(zx)y &= \left|\begin{array}{cccc}\mathsf{y}&\mathsf{y}&\mathsf{y}&\mathsf{n}\\ \mathsf{n}&\mathsf{y}&\mathsf{n}&\mathsf{y}\\ \mathsf{n}&\mathsf{n}&\mathsf{y}&\mathsf{n}\end{array}\right| \mathsf{I}\ ([x,y]\ y\ x)\ ([x,z]\ z\ x)\ \mathsf{I}\\ &= \left|\begin{array}{cccc}\mathsf{y}&\mathsf{y}&\mathsf{y}&\mathsf{n}\\ \mathsf{n}&\mathsf{y}&\mathsf{n}&\mathsf{y}\\ \mathsf{n}&\mathsf{n}&\mathsf{y}&\mathsf{n}\end{array}\right| \mathsf{I}\ \left(\left|\begin{array}{cc}\mathsf{n}&\mathsf{y}\\ \mathsf{y}&\mathsf{n}\end{array}\right| \mathsf{I}\ \mathsf{I}\right)\ \left(\left|\begin{array}{cc}\mathsf{n}&\mathsf{y}\\ \mathsf{y}&\mathsf{n}\end{array}\right| \mathsf{I}\ \mathsf{I}\right)\ \mathsf{I}.\end{aligned}$$

The top row **yyyn** of the 3×4 array representation that occurs in this example shows the pattern of occurrences of the variable x in the primal components of the input term. Similarly, the second row, namely **nyny**, shows the pattern of occurrences of the variable y in the primal components of the input term and the third row does the same for the variable z.

5 Properties

5.1 Structure-preserving

One of the useful properties that algorithm (L) possesses is that it is structure-preserving. This is a generalisation of one of the four properties that Turner [7] says that an abstraction algorithm should have if it is to be any good when used to implement a functional language. He says that the algorithm should produce short abstracts, use only a finite number of combinators, be uni-variate, but well-behaved under self-composition. To understand the last requirement, let P_1 and P_2 be terms of combinatory logic such that each of the variables $x_1, x_2, \ldots, x_a$ occurs in both of them. Furthermore, let $T_1 = [x_1]\ P_1\ P_2$, $T_2 = [x_2]\ T_1$, $T_3 = [x_3]\ T_2$, $\ldots$, $T_a = [x_a]\ T_{a-1}$. Then the algorithm used to produce $[x]\ X$ is *well-behaved under self-composition* if T_i, for $1 \le i \le a$, has the form $K\ Q_1\ Q_2$, where K is a term consisting entirely of combinators. The property of being structure-preserving is a generalisation of Turner's property of being well-behaved under self-composition. Let $P_1, P_2, \ldots, P_m$ be m terms. Any of the variables $x_1, x_2, \ldots, x_a$ can occur in any of these terms, but none of them has to. Furthermore, let $T_1 = [x_1]\ P_1\ P_2\ \ldots\ P_m$ and T_i, for $2 \le i \le a$, be as defined above. Then an algorithm that produces abstracts T_i, for $1 \le i \le a$, of the form $K\ Q_1\ Q_2\ \ldots\ Q_m$, where K is a term consisting entirely of combinators, is said to be *structure-preserving.* Elsewhere [3, pp. 4–5] I show that Turner's algorithm is not structure-preserving in this sense. In fact, no algorithm which uses only a finite number of combinators can be. Algorithm (L), however, is structure-preserving. (The reason why algorithm (L) does not contain the clauses $[x]\ x = \mathsf{I}$ and $[x]\ E\ x = E$, when x does not occur in E, is because the presence of those clauses would mean that the resulting algorithm is not structure-preserving.)

Algorithm (M) can also be said to be structure-preserving, though in a sense which is analogous to that in which algorithm (L) is said to be structure-preserving. To be more precise, let $P = P_1\ P_2\ \ldots\ P_m$ and $[\vec{x}^k] = [x_1^k][x_2^k]\ldots[x_{a_k}^k]$. Furthermore, let $T_1 = [\vec{x}^1]\ P$, $T_2 = [\vec{x}^2]\ T_1$, $T_3 = [\vec{x}^3]\ T_2$, $\ldots$, $T_k = [\vec{x}^k]\ T_{k-1}$. Then each term T_i, $1 \le i \le k$, is of the form $K\ Q_1\ Q_2\ \ldots\ Q_m$, where K is made up entirely of combinators. (The reason why algorithm (M) does not contain the clause $[\vec{x}]\ E\ \vec{x} = E$, where E does not contain any of the variables in $\vec{x}$, is because the presence of that clause would mean that the resulting algorithm would no longer be structure-preserving.)

5.2 Permutation of Primal Components

In addition to being structure-preserving, algorithm (L) also has a number of further interesting properties. In this subsection I consider the following property: given the result of applying algorithm (L) to a term P, we can easily obtain the result of applying it to a term obtained from P be permuting all its primal components except the first. We do this by applying the same permutation to the letters making up the yes-no representation, except the first, and the primal components making up the abstract, except the first two. (Note that the first primal component of the abstract is a yes-no representation. Note also that, as here used, a permutation is just a bijection of a set to itself.) An example should clarify this property. The following is an example of the use of algorithm (L):

$$\begin{aligned}[x]\ x\ (y\ z)\ (x\ y)\ z\ (y\ x\ z) &= \textsf{\textbf{ynyny I}}\ (y\ z)\ ([x]\ x\ y)\ z\ ([x]\ y\ x\ z)\\ &= \textsf{\textbf{ynyny I}}\ (y\ z)\ (\textsf{\textbf{yn I}}\ y)\ z\ (\textsf{\textbf{nyn}}\ y\ \textsf{\textbf{I}}\ z).\end{aligned}$$

If we now want to know what $[x]\ x\ z\ (y\ x\ z)\ (x\ y)\ (y\ z)$, say, is, we do not have to apply algorithm (L). We can instead read off the result from the above abstract. This is because the second input term is obtained from the first by applying a permutation to its primal components, except the first. Thus,

$$[x]\ x\ z\ (y\ x\ z)\ (x\ y)\ (y\ z) = \textsf{\textbf{ynyyn I}}\ z\ (\textsf{\textbf{nyn}}\ y\ \textsf{\textbf{I}}\ z)\ (\textsf{\textbf{yn I}}\ y)\ (y\ z).$$

In this case the permutation is the bijection $\{\langle 2,4\rangle, \langle 3,5\rangle, \langle 4,3\rangle, \langle 5,2\rangle\}$. Note that the non-first primal components of the non-first primal components of the input term can themselves be permuted, if they contain the abstraction variable, by means of a possibly different permutation, and so on until we reach primal components which are atoms. For example, if we are interested in working out $[x]\ x\ z\ (y\ z\ x)\ (x\ y)\ (y\ z)$, we do not have to apply algorithm (L) as the result can be read off from the first abstract given above. The result is $\textsf{\textbf{ynyyn I}}\ z\ (\textsf{\textbf{nny}}\ y\ z\ \textsf{\textbf{I}})\ (\textsf{\textbf{yn I}}\ y)\ (y\ z)$. These observations are made precise in the following proposition.

Proposition 1 *Let σ be a permutation of $\{2, 3, \ldots, m\}$ and let $P = P_1\ P_2\ \ldots\ P_m$, where P_1 is an atom. Then if*

$$[x]\ P = \beta_1\beta_2\ldots\beta_m\ Q_1\ Q_2\ \ldots\ Q_m,$$

where $\beta_1\beta_2\ldots\beta_m$ and the Q_i are as specified by algorithm (L), then

$$[x]\ P_1\ P_{\sigma(2)}\ \ldots\ P_{\sigma(m)} = \beta_1\beta_{\sigma(2)}\ldots\beta_{\sigma(m)}\ Q_1\ Q_{\sigma(2)}\ \ldots\ Q_{\sigma(m)}.$$

Proof The proof is by induction on $rpv(\{x\}, P)$. In the base case $rpv(\{x\}, P) = 0$. When that happens let us assume that

$$[x]\ P = \beta_1\beta_2\ldots\beta_m\ Q_1\ Q_2\ \ldots\ Q_m,$$

where $\beta_1\beta_2\ldots\beta_m$ and the Q_j are as specified by algorithm (L). Because $rpv(\{x\}, P) = 0$, the algorithm is not applied recursively. Therefore, either $Q_i = P_i$, when $x \notin FV(P_i)$, or $Q_i = \textsf{\textbf{I}}$, when $x = P_i$. On this assumption we also have that

$$[x]\ P_1\ P_{\sigma(2)}\ \ldots\ P_{\sigma(m)} = \gamma_1\gamma_2\ldots\gamma_m\ R_1\ R_2\ \ldots\ R_m.$$

where $\gamma_1\gamma_2\ldots\gamma_m$ and the R_j are as specified by algorithm (L). P_1 is an atom, so $R_1 = P_1$, if $P_1 \neq x$, and $R_i = \textsf{\textbf{I}}$, if $P_1 = x$. Furthermore, for $2 \leq i \leq m$,

$$\gamma_i = \begin{cases} \textsf{\textbf{y}}, & \text{if } x = P_{\sigma(i)},\\ \textsf{\textbf{n}}, & \text{otherwise;}\end{cases}$$

and, for $2 \leq i \leq m$, $R_i = P_{\sigma(i)}$, if $x \notin FV(P_{\sigma(i)})$, and $R_i = \textsf{\textbf{I}}$, if $x = P_{\sigma(i)}$. Thus, $\gamma_1 = \beta_1$ and $R_1 = P_1$ and, for $2 \leq i \leq m$, $\gamma_i = \beta_{\sigma(i)}$ and $R_i = Q_{\sigma(i)}$. Thus, using conditionalisation, the base case is established.

In the inductive step $rpv(\{x\}, P) \neq 0$. Let us assume that

$$[x]\ P = \beta_1\beta_2 \dots \beta_m\ Q_1\ Q_2\ \dots\ Q_m,$$

where $\beta_1\beta_2\dots\beta_m$ and the Q_j are as specified by algorithm (L). On this assumption we also have that

$$[x]\ P_1\ P_{\sigma(2)}\ \dots\ P_{\sigma(m)} = \gamma_1\gamma_2\dots\gamma_m\ R_1\ R_2\ \dots\ R_m.$$

where $\gamma_1\gamma_2\dots\gamma_m$ and the R_j are as specified by algorithm (L). Since P_1 is the first primal component of P, $\beta_1 = \gamma_1$ and $R_1 = Q_1$. For $2 \leq i \leq m$, we have that

$$\gamma_i = \begin{cases} \mathsf{y}, & \text{if } x \in FV(P_{\sigma(i)}), \\ \mathsf{n}, & \text{otherwise;} \end{cases}$$

and also that

$$R_i = \begin{cases} \mathsf{I}, & \text{if } P_{\sigma(i)} = x, \\ [x]\ P_{\sigma(i)}, & \text{if } P_{\sigma(i)} \neq x, \text{ but } x \in FV(P_{\sigma(i)}), \\ P_{\sigma(i)}, & \text{if } x \notin FV(P_{\sigma(i)}). \end{cases}$$

When $x = P_{\sigma(i)}$, then $\gamma_i = \beta_{\sigma(i)}$ and $R_i = Q_{\sigma(i)}$. When $x \notin FV(P_{\sigma(i)})$, then $\gamma_i = \beta_{\sigma(i)}$ and $R_i = Q_{\sigma(i)}$. When $x \neq P_{\sigma(i)}$ and $x \in FV(P_{\sigma(i)})$, then $\gamma_i = \beta_{\sigma(i)}$ and $R_i = Q_{\sigma(i)}$, by the inductive hypothesis, as $rpv(\{x\}, P_{\sigma(i)}) < rpv(\{x\}, P)$. Thus, the inductive step is established by the use of conditionalisation and the result follows by induction. QED.

5.3 Permutation of Abstractions

An unusual property of algorithm (L) emerges when it is used several times in order to abstract different variables from a term, for example, if it is used to produce $[z]\ ([y]\ ([x]\ x\ (y\ z)\ (y\ x\ z)))$, say. If we now want to know what the abstract would be if the variables were abstracted in a different order, then that can be read off from the initial abstract. In other words, it is straightforward to produce, say, $[y]\ ([x]\ ([z]\ x\ (y\ z)\ (y\ x\ z)))$. All that is required is to permute various parts of the abstract in the same way that the abstraction variables have been permuted, as the following shows:

$$\begin{aligned} & [z]\ ([y]\ ([x]\ x\ (y\ z)\ (y\ x\ z))) \\ = {} & \mathsf{nn}\underbrace{\mathsf{nyy}}_{\beta_z}\ \mathsf{n}\underbrace{\mathsf{nyy}}_{\beta_y}\ \underbrace{\mathsf{yny}}_{\beta_x}\ \mathsf{I}\ (\mathsf{n}\underbrace{\mathsf{ny}}_{\gamma_z}\ \underbrace{\mathsf{yn}}_{\gamma_y}\ \mathsf{I}\ \mathsf{I})\ (\mathsf{nn}\underbrace{\mathsf{nny}}_{\delta_z}\ \underbrace{\mathsf{nynn}}_{\delta_y}\ \underbrace{\mathsf{nyn}}_{\delta_x}\ \mathsf{I}\ \mathsf{I}\ \mathsf{I}). \\ & [y]\ ([x]\ ([z]\ x\ (y\ z)\ (y\ x\ z))) \\ = {} & \mathsf{nn}\underbrace{\mathsf{nyy}}_{\beta_y}\ \mathsf{n}\underbrace{\mathsf{yny}}_{\beta_x}\ \underbrace{\mathsf{nyy}}_{\beta_z}\ \mathsf{I}\ (\mathsf{n}\underbrace{\mathsf{yn}}_{\gamma_y}\ \underbrace{\mathsf{ny}}_{\gamma_z}\ \mathsf{I}\ \mathsf{I})\ (\mathsf{nn}\underbrace{\mathsf{ynn}}_{\delta_y}\ \mathsf{n}\underbrace{\mathsf{nyn}}_{\delta_x}\ \underbrace{\mathsf{nny}}_{\delta_z}\ \mathsf{I}\ \mathsf{I}\ \mathsf{I}). \end{aligned}$$

These observations are made precise in the following proposition.

Proposition 2 *Let us assume the following:*

$$\begin{aligned} & [x_1]\ ([x_2]\ (\dots([x_a]\ P_1\ P_2\ \dots\ P_m)\dots)) \\ = {} & \underbrace{\mathsf{nn}\dots\mathsf{n}}_{a-1\ \text{times}}\beta_{1,1}\beta_{1,2}\dots\beta_{1,m}\ \ \underbrace{\mathsf{nn}\dots\mathsf{n}}_{a-2\ \text{times}}\beta_{2,1}\beta_{2,2}\dots\beta_{2,m}\ \dots\ \beta_{a,1}\beta_{a,2}\dots\beta_{a,m}\ Q_1\ Q_2\ \dots\ Q_m \end{aligned}$$

where, for $1 \leq i \leq a$ and $1 \leq j \leq m$,

$$\beta_{i,j} = \begin{cases} \mathsf{y}, & \textit{if } x_i \in FV(P_j), \\ \mathsf{n}, & \textit{otherwise;} \end{cases}$$

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

$$Q_j = \begin{cases} \mathsf{I}, & \textit{if } P_j = x_i,\ \exists i \in 1..a, \\ P_j, & \textit{if } x_i \notin FV(P_j),\ \forall i \in 1..a, \\ [x_{j_1}]([x_{j_2}](\ldots([x_{j_{f(j)}}]\ P_j)\ldots)), & \textit{otherwise,} \end{cases}$$

where $f(j)$ is the number of the variables $x_1, x_2, \ldots, x_a$ that occur in P_j and each of the variables $x_{j_1}, x_{j_2}, \ldots, x_{j_{f(j)}}$, where $j_1 < j_2 < \ldots < j_{f(j)}$, occur in P_j and no other variable x_i, for $1 \leq i \leq a$, does. Then, where σ is a permutation of the numbers $\{1, 2, \ldots, a\}$,

$$\begin{aligned} &[x_{\sigma(1)}]\ ([x_{\sigma(2)}]\ (\ldots([x_{\sigma(a)}]\ P_1\ P_2\ \ldots\ P_m)\ldots)) \\ =\ & \underbrace{\mathsf{nn}\ldots\mathsf{n}}_{a-1 \text{ times}}\beta_{\sigma(1),1}\beta_{\sigma(1),2}\ldots\beta_{\sigma(1),m}\ \underbrace{\mathsf{nn}\ldots\mathsf{n}}_{a-2 \text{ times}}\beta_{\sigma(2),1}\beta_{\sigma(2),2}\ldots\beta_{\sigma(2),m}\ \cdots \\ & \quad \beta_{\sigma(a),1}\beta_{\sigma(a),2}\ldots\beta_{\sigma(a),m}\ R_1\ R_2\ \ldots\ R_m \end{aligned}$$

where the $\beta_{i,j}$, for $1 \leq i \leq a$ and $1 \leq j \leq m$, are as defined above and, for $1 \leq j \leq m$,

$$R_j = \begin{cases} \mathsf{I}, & \textit{if } P_j = x_{\sigma(i)},\ \exists i \in 1..a, \\ P_j, & \textit{if } x_{\sigma(i)} \notin FV(P_j),\ \forall i \in 1..a, \\ [x_{\sigma(j_1)}]([x_{\sigma(j_2)}](\ldots([x_{\sigma(j_{f(j)})}]\ P_j)\ldots)), & \textit{otherwise.} \end{cases}$$

Proof Let $P = P_1\ P_2\ \ldots\ P_m$. The proof of this result is by induction on the value of $\phi(a, \mathit{rpv}(\{x_1, x_2, \ldots, x_a\}, P))$, where $\phi : \mathbf{N}_1 \times \mathbf{N} \to \mathbf{N}_1$ is a total bijection. ($\mathbf{N}$ is the set of all non-negative whole numbers and $\mathbf{N}_1$ is the set of all positive whole numbers.) The function ϕ can be defined as follows:

$$\phi(p, q) = \begin{cases} p^2, & p = q + 1; \\ (p-1)^2 + q + 1, & p > q + 1; \\ q^2 + q + p, & p < q + 1. \end{cases}$$

In the base case $\phi(a, \mathit{rpv}(\{x_1, x_2, \ldots, x_a\}, P)) = 1$. Thus, $a = 1$ and $\mathit{rpv}(\{x_1, x_2, \ldots, x_a\}, P) = 0$. In this case we have that

$$[x_1]\ P_1\ P_2\ \ldots\ P_m = \beta_{1,1}\beta_{1,2}\ldots\beta_{1,m}\ Q_1\ Q_2\ \ldots\ Q_m,$$

where $\beta_{1,1}\beta_{1,2}\ldots\beta_{1,m}$ and the Q_j are as specified in the statement of the assumption of the proposition above. We also have that

$$[x_{\sigma(1)}]\ P_1\ P_2\ \ldots\ P_m = \gamma_{1,1}\gamma_{1,2}\ldots\gamma_{1,m}\ R_1\ R_2\ \ldots\ R_m,$$

where $\gamma_{1,1}\gamma_{1,2}\ldots\gamma_{1,m}$ and the R_j are as specified in the statement of the assumption of the proposition above. From this we need to show that the then-part of the proposition holds in the base case. As $\mathit{rpv}(\{x_1, x_2, \ldots, x_a\}, P) = 0$, none of the primal components of P which are combinations contain x_1. Thus, the algorithm is not applied recursively. There is only one permutation of the number 1, namely the identity function. Therefore, $\gamma_{1,1}\gamma_{1,2}\ldots\gamma_{1,m} = \beta_{\sigma(1),1}\beta_{\sigma(1),2}\ldots\beta_{\sigma(1),m}$ and $R_i = Q_i$, for $1 \leq i \leq m$. Thus, the base case has been established.

In the inductive step $\phi(a, \mathit{rpv}(\{x_1, x_2, \ldots, x_a\}, P)) \neq 1$. We then have that

$$\begin{aligned} &[x_1]\ ([x_2]\ (\ldots([x_a]\ P_1\ P_2\ \ldots\ P_m)\ldots)) \\ =\ & \underbrace{\mathsf{nn}\ldots\mathsf{n}}_{a-1 \text{ times}}\beta_{1,1}\beta_{1,2}\ldots\beta_{1,m}\ \underbrace{\mathsf{nn}\ldots\mathsf{n}}_{a-2 \text{ times}}\beta_{2,1}\beta_{2,2}\ldots\beta_{2,m}\ \ldots\ \beta_{a,1}\beta_{a,2}\ldots\beta_{a,m}\ Q_1\ Q_2\ \ldots\ Q_m \end{aligned}$$

where the $\beta_{i,j}$ and the Q_j are as specified in the statement of the assumption part of the proposition. Let σ be a permutation of the numbers $1, 2, 3, \ldots, a$. Then we also have that

$$\begin{aligned}
&[x_{\sigma(1)}]\ ([x_{\sigma(2)}]\ (\ldots([x_{\sigma(a)}]\ P_1\ P_2\ \ldots\ P_m)\ldots))\\
=\ &\underbrace{\mathbf{nn}\ldots\mathbf{n}}_{a-1\text{ times}}\gamma_{1,1}\gamma_{1,2}\cdots\gamma_{1,m}\ \underbrace{\mathbf{nn}\ldots\mathbf{n}}_{a-2\text{ times}}\gamma_{2,1}\gamma_{2,2}\cdots\gamma_{2,m}\ \cdots\ \gamma_{a,1}\gamma_{a,2}\cdots\gamma_{a,m}\\
&R_1\ R_2\ \ldots\ R_m
\end{aligned}$$

where the $\gamma_{i,j}$ and the R_j are as specified in the statement of the assumption part of the proposition. On this basis we need to show that the then-part of the proposition obtains in the inductive step of the proof. Because the permutation σ has been applied to the indices of the abstracting variables, $\gamma_{i,j} = \beta_{\sigma(i),j}$, for $1 \le i \le a$ and $1 \le j \le m$.

When $P_j = x_i$, for some i such that $1 \le i \le a$, then $Q_j = \mathbf{I}$ and $R_j = \mathbf{I}$. Therefore, $R_j = Q_j$.

When $x_i \notin FV(P_j)$, for any i such that $1 \le i \le a$, then $Q_j = P_j$ and $R_j = P_j$. Therefore, $R_j = Q_j$.

When $P_j \ne x_i$, for any i such that $1 \le i \le a$, and $x_i \in FV(P_j)$, for some i such that $1 \le i \le a$, then

$$Q_j = [x_{j_1}]\ ([x_{j_2}]\ (\ldots([x_{j_{f(j)}}]\ P_j)\ldots)).$$

In this situation, by the inductive hypothesis, since $\phi(f(j), rpv(\{x_{j_1}, x_{j_2}, \ldots, x_{j_{f(j)}}\}, P_j)) < \phi(a, rpv(\{x_1, x_2, \ldots, x_a\}, P))$, it follows that

$$R_j = [x_{\sigma(j_1)}]\ ([x_{\sigma(j_2)}]\ (\ldots([x_{\sigma(j_{f(j)})}]\ P_j)\ldots)).$$

Thus, the inductive step has been established and the result follows by induction. QED.

5.4 Relating Yes-no and Array Representations

There is an interesting connection between the yes-no representations produced when algorithm (L) is used to abstract different variables from the same input term and the array representation produced when algorithm (M) is used to abstract those variables in a single sweep from the same input term. The following example illustrates this connection. Let $P = y\ (x\ z)\ (y\ x\ z)$. Then we have

$$\begin{aligned}
[x]\ P &= \mathbf{nyy}\ y\ (\mathbf{yn}\ \mathbf{I}\ z)\ (\mathbf{nyn}\ y\ \mathbf{I}\ z).\\
[y]\ P &= \mathbf{yny}\ \mathbf{I}\ (x\ z)\ (\mathbf{ynn}\ \mathbf{I}\ x\ z).\\
[z]\ P &= \mathbf{nyy}\ y\ (\mathbf{ny}\ x\ \mathbf{I})\ (\mathbf{nny}\ y\ x\ \mathbf{I}).
\end{aligned}$$

We also have that

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

If we look at the yes-no representations that are the first primal components of the three abstracts produced by $[x]\ P$, $[y]\ P$ and $[z]\ P$, we notice that they are the same as the three rows of the array representation which is the first primal component of the abstract produced by $[x, y, z]\ P$. (There are also connections between the other representations produced, but they are not my concern here.) This connection can be stated more precisely as follows. Let

$P = P_1\ P_2\ \ldots\ P_m$. Then we have

$$\begin{aligned}
[x_1]\ P &= \beta_{1,1}\beta_{1,2}\ldots\beta_{1,m}\ Q^1_1\ Q^1_2\ \ldots\ Q^1_m,\\
[x_2]\ P &= \beta_{2,1}\beta_{2,2}\ldots\beta_{2,m}\ Q^2_1\ Q^2_2\ \ldots\ Q^2_m,\\
&\ldots\\
[x_a]\ P &= \beta_{a,1}\beta_{a,2}\ldots\beta_{a,m}\ Q^a_1\ Q^a_2\ \ldots\ Q^a_m,
\end{aligned}$$

where the yes-no representations 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 γ and the Q_j, for $1 \leq j \leq m$ are as specified by algorithm (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 my concern here.) This connection is a straightforward consequence of the definitions of algorithms (L) and (M).

5.5 Properties of Array Representations

No operations suggested by standard matrix operations have been found to have analogues in array representations, but some other operations have already been discovered. For example, if α is an $a \times (m+1)$ array representation such that $\alpha_{i,1} = \mathbf{n}$, for $1 \leq i \leq a$, and γ is a $b \times m$ array representation, then $\alpha\ \gamma =_{\beta\eta} \delta$, where $=_{\beta\eta}$ is $\beta\eta$ equality and δ is an $(a+b) \times m$ array representation such that, for $1 \leq i \leq a$ and $1 \leq j \leq m$, $\delta_{i,j} = \alpha_{i,j+1}$ and $\delta_{a+i,j} = \gamma_{i,j}$. (The proof of this result is a straightforward, though tedious, consequence of the definition of reduction for array representations given in Fig. 2.) For example,

$$\left|\begin{array}{ccccccc}
\mathbf{n} & \mathbf{n} & \mathbf{n} & \mathbf{y} & \mathbf{n} & \mathbf{n} & \mathbf{n}\\
\mathbf{n} & \mathbf{n} & \mathbf{n} & \mathbf{n} & \mathbf{n} & \mathbf{n} & \mathbf{y}
\end{array}\right|
\left|\begin{array}{cccccc}
\mathbf{y} & \mathbf{n} & \mathbf{n} & \mathbf{y} & \mathbf{n} & \mathbf{n}\\
\mathbf{n} & \mathbf{y} & \mathbf{n} & \mathbf{n} & \mathbf{y} & \mathbf{n}\\
\mathbf{y} & \mathbf{y} & \mathbf{y} & \mathbf{y} & \mathbf{n} & \mathbf{y}
\end{array}\right|
=_{\beta\eta}
\left|\begin{array}{cccccc}
\mathbf{n} & \mathbf{n} & \mathbf{y} & \mathbf{n} & \mathbf{n} & \mathbf{n}\\
\mathbf{n} & \mathbf{n} & \mathbf{n} & \mathbf{n} & \mathbf{n} & \mathbf{y}\\
\mathbf{y} & \mathbf{n} & \mathbf{n} & \mathbf{y} & \mathbf{n} & \mathbf{n}\\
\mathbf{n} & \mathbf{y} & \mathbf{n} & \mathbf{n} & \mathbf{y} & \mathbf{n}\\
\mathbf{y} & \mathbf{y} & \mathbf{y} & \mathbf{y} & \mathbf{n} & \mathbf{y}
\end{array}\right|.$$

The significance of this property is that we can "partition" a multi-variate abstraction performed by algorithm (M), because the following holds:

$$[x_1, x_2, \ldots, x_a]\ ([x_{a+1}, x_{a+2}, \ldots, x_{a+b}]\ P) =_{\beta\eta} [x_1, x_2, \ldots, x_b]\ P.$$

In fact, the "partitioning" does not have to be into two. It can be into any number of component abstractions. Such different "partitionings" would give rise to different relations between array representations.

6 Conclusion

Although the original motivation for developing iconic representations and their associated abstraction algorithms was that of improving the way in which pure functional languages are implemented, I have shown in this paper that algorithms (L) and (M) have several surprising and unusual properties. In addition, I have shown that there is an interesting connection between the representations produced by these two algorithms and that a useful connection between certain array representations can be established which allows us to "partition" multi-variate abstraction. I hope that these properties and connections stimulate other people to investigate these algorithms and the representations that they use. I find them fascinating and I am confident that many more interesting and useful properties that they possess are waiting to be discovered.

Acknowledgements

I would like to thank David Stevens for many useful conversations about bracket abstraction and Donald Peterson for introducing me to the idea of rerepresentation and for showing me its importance. I am grateful to Debra Barton for reading through a draft of this paper and for making a number of helpful suggestions for improving it.

References

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

[2] Antoni Diller. Efficient multi-variate abstraction using an array representation for combinators. Research Report CSR–99–13, School of Computer Science, University of Birmingham, December 1999.

[3] Antoni Diller. Making abstraction behave by rerepresenting combinators. Research Report CSR–99–12, School of Computer Science, University of Birmingham, November 1999.

[4] 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.

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

[6] 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.

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