Information Processing Letters 84 (2002) 311–317

Information Processing Letters

www.elsevier.com/locate/ipl

Efficient multi-variate abstraction using an array representation for combinators

Antoni Diller

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

Received 15 February 2000; received in revised form 26 April 2002

Communicated by F.B. Schneider

Keywords: Bracket abstraction; Combinatory logic; Compilers; Functional programming

1. Introduction

Turner [9] showed how a pure functional programming language could be implemented using combinatory logic in a practicable way. This method is currently out of fashion, but for some time I have believed that its full potential is still to be realized. This belief was partially vindicated by Stevens [7] who developed a family of interesting algorithms which use a novel notation for combinators. The work reported here was inspired by that of Stevens, but is significantly different. (Additional information about the motivation behind the current research can be found elsewhere [2, pp. 2–5].)

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 [4].

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 or superscripts, are used for variables. A *term* is defined thus:

(a) Every variable is a term;
(b) Every constant is a term;
(c) If P and Q are terms, then so is (PQ).

The letters P, Q, R, S, T and X, sometimes decorated with subscripts or superscripts, are used for terms. An *atom* is a variable or a constant. 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.

A *subterm* is defined thus: P is a subterm of P; 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 $P_1 P_2 \dots P_m$, where P_1 is an atom and $m \geqslant 1$. The P_i are know as the *primal components* of P. The non-standard notion of a *subprimal component*

E-mail address: a.r.diller@cs.bham.ac.uk (A. Diller).

PII: S0020-0190(02)00318-6

is defined thus: P is a subprimal component of P; 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.

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. *Substituting* P for every occurrence of x in X, written $[P/x]X$, is defined thus:

(a) $[P/x]x \equiv x$;
(b) $[P/x]Y \equiv Y$, if Y is an atom distinct from x;
(c) $[P/x]QR = ([P/x]Q)([P/x]R)$.

A term of the form $\mathsf{K}PQ$ or $\mathsf{S}PQR$ is a *redex*. *Contracting* an instance of a redex in a term S means replacing one occurrence of $\mathsf{K}PQ$ by P or one occurrence of $\mathsf{S}PQR$ 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*. 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 B, C and I can be defined in terms of K and S:

$$\mathsf{B} \triangleq \mathsf{S(KS)K}, \quad \mathsf{C} \triangleq \mathsf{S(BBS)(KK)} \quad \text{and} \quad \mathsf{I} \triangleq \mathsf{SKK}.$$

These contract thus:

$$\mathsf{B}PQR \to P(QR),$$

$$\mathsf{C}PQR \to PRQ \quad \text{and} \quad \mathsf{I}P \to P.$$

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 $[x]X \equiv Q$, then X is the *input term* and Q the *abstract*. Usually, in combinatory logic, *multi-variate abstraction* $[x_1, x_2, \ldots, x_a]X$ is defined to be the same as $[x_1]([x_2](\ldots([x_{a-1}]([x_a]X))\ldots))$. In this paper, however, it represents an operation that removes several variables simultaneously. Furthermore, all the variables in the bracket prefix $[x_1, x_2, \ldots, x_a]$ are assumed to be distinct.

Two non-standard notations for combinators are introduced here, namely as strings of the letters y and n, called yn-*strings*, and as arrays or matrices of the letters y and n, called yn-*arrays*. The letters γ and δ are used for arbitrary yn-arrays and β for a yn-string. *size*(β) is the number of occurrences of y and n in β and β_i, for $1 \leqslant i \leqslant$ *size*(β), is the ith letter in β. String concatenation is represented by juxtaposition. If γ is an $a \times m$ yn-array, then $\gamma_{i,j}$, for $1 \leqslant i \leqslant a$ and $1 \leqslant j \leqslant m$, is the jth letter in row i. Note that if β is a yn-string, it is assumed that $\#\beta = 1$. Similarly, if γ is an yn-array, it is assumed that $\#\gamma = 1$, but this assumption is discussed in the conclusion.

Let β be a yn-string. Then a β-*redex* is any term of the form $\beta P_1 P_2 \ldots P_{m+1}$, where $m = \mathit{size}(\beta)$. *Contracting* an instance of a β-redex in a term S means replacing one occurrence of $\beta P_1 P_2 \ldots P_{m+1}$ by $Q_1 Q_2 \ldots Q_m$, where, for $1 \leqslant i \leqslant m$,

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

Let $[\vec{x}] = [x_1, x_2, \ldots, x_a]$. Then $rpv([\vec{x}], P)$ is the number of distinct non-atomic subprimal components of P, other than P itself, which contain at least one of the variables $x_1, x_2, \ldots, x_a$. There is an alternative characterization of rpv. Let P be represented using the fewest possible parentheses. Then $rpv([\vec{x}], P)$ is half the number of parentheses that enclose subterms containing at least one of the variables $x_1, x_2, \ldots, x_a$. Thus,

$$rpv\big([x, y, z], x(yz)(wv)z\big) = 1 \quad \text{and}$$
$$rpv\big([x, y], x(y(wv))(w(xv))\big) = 3.$$

Let $P \equiv P_1 P_2 \ldots P_m$, where P_1 is an atom. Then

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

where $(\forall i \in 1..a)$ means ‘for all i such that $1 \leqslant i \leqslant a$’ and $(\exists i \in 1..a)$ means ‘for some i such that $1 \leqslant i \leqslant a$’. Putting a conditional inside a summation may be unusual, but its meaning is straightforward. Let $\Gamma(j)$ be a Boolean-valued function and let $f(j)$ and $g(j)$ be integer-valued ones. Then

$$\sum_{j=1}^{m} \textbf{if } \Gamma(j) \textbf{ then } f(j) \textbf{ else } g(j)$$
$$= \big(\textbf{if } \Gamma(1) \textbf{ then } f(1) \textbf{ else } g(1)\big) + \big(\textbf{if } \Gamma(2) \textbf{ then } f(2) \textbf{ else } g(2)\big) + \cdots + \big(\textbf{if } \Gamma(m) \textbf{ then } f(m) \textbf{ else } g(m)\big).$$

3. Contraction

In order to explain how yn-arrays are contracted two functions have to be defined on yn-strings: $yc(\beta)$ 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 $1 \leqslant i \leqslant yc(\beta)$; if $i > yc(\beta)$, then $posy(i,\beta)$ is not defined. For example, $yc(\mathsf{ynnynyy}) = 4$, $posy(1,\mathsf{ynnynyy}) = 1$ and $posy(2,\mathsf{ynnynyy}) = 4$. Let γ be an $a \times m$ yn-array. Then it contracts thus:

$$\gamma P_1 P_2 \ldots P_m P_{m+1} \ldots P_{m+a} \to_1 Q_1 Q_2 \ldots Q_m,$$

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

$$\begin{vmatrix} \mathsf{n} & \mathsf{n} & \mathsf{y} & \mathsf{n} \\ \mathsf{y} & \mathsf{n} & \mathsf{y} & \mathsf{n} \\ \mathsf{n} & \mathsf{n} & \mathsf{y} & \mathsf{y} \end{vmatrix} P_1 P_2 P_3 P_4 P_5 P_6 P_7$$
$$\to_1 P_1 P_6 P_2 (P_3 P_5 P_6 P_7)(P_4 P_7).$$

An $a \times m$ yn-array γ has to be followed by at least $a + m$ terms in order for a contraction to be possible. Informally, the first m terms following γ can be thought of as functions and the next a can be seen as their possible arguments. The yn-array tells us which of these are going to be passed to the functions to become their actual arguments. Thus, the jth column tells us which of $P_{m+1}, P_{m+2}, \ldots, P_{m+a}$ follow P_j in the contractum of the yn-array: P_{m+i} only occurs if $\gamma_{i,j}$ is y. For example, the 4th column of the yn-array used in the example is nny. This tells us which of the terms P_5, P_6 and P_7 follow P_4 in the contractum. As only the last letter of nny is y, only P_7 does. This means that $(P_4 P_7)$ occurs in the contractum.

4. Translation

Many people, on first encountering yn-strings, think that they are similar to director strings [6]. Director strings, however, are not combinators. They are elements of a new formal system called 'the director string calculus' whose properties have to be established from scratch. yn-strings and yn-arrays, by contrast, are just alternative notations for combinators, just as Roman and Arabic numerals are alternative representations for numbers. This can be established by translating them into the usual notation for combinators. This is achieved by the function *Trans* which employs *trans* which translates yn-strings into standard combinators. The function *trans* uses the series of combinators B_i, for $i \geqslant 1$, defined thus:

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

Note that each B_i, for $i \geqslant 1$, has the same effect as the B^i defined in [1, pp. 163–164]. The function *trans* is defined thus:

$$\begin{aligned} &\mathit{trans}(\mathsf{y}) \triangleq \mathsf{BI}, \\ &\mathit{trans}(\mathsf{n}) \triangleq \mathsf{K}, \\ &\mathit{trans}(\beta\mathsf{y}) \triangleq \mathsf{B}_i \mathsf{S}\, \mathit{trans}(\beta), \quad \text{if } \mathit{size}(\beta) \geqslant 1, \\ &\mathit{trans}(\beta\mathsf{n}) \triangleq \mathsf{B}_i \mathsf{C}\, \mathit{trans}(\beta), \quad \text{if } \mathit{size}(\beta) \geqslant 1, \end{aligned}$$

where $i = \mathit{size}(\beta)$. For example,

$$\begin{aligned} \mathit{trans}(\mathsf{nny}) &= \mathsf{B}_2 \mathsf{S}\, \mathit{trans}(\mathsf{nn}) \\ &= \mathsf{B}_2\, \mathsf{S}\, \big(\mathsf{B}_1\, \mathsf{C}\, \mathit{trans}(\mathsf{n})\big) \\ &= \mathsf{B}_2\, \mathsf{S}\, (\mathsf{B}_1\, \mathsf{C}\, \mathsf{K}). \end{aligned}$$

The function *trans* is *correct* if $\mathit{trans}(\beta)\ P_1 P_2 \ldots P_m P_{m+1} \to Q$, where $m = \mathit{size}(\beta)$ and Q is the result of contracting $\beta P_1 P_2 \ldots P_m P_{m+1}$. That *trans* is correct is proved elsewhere [2, pp. 9–10].

The function *Trans* is defined thus:

$$\begin{aligned} \mathit{Trans}(\gamma) \triangleq\ & \overbrace{\mathit{trans}(\underbrace{\mathsf{nn}\ldots\mathsf{n}}_{a-1 \text{ times}}\ \gamma_{1,1}\gamma_{1,2}\ldots\gamma_{1,m})}^{t_a} \\ & \overbrace{\mathit{trans}(\underbrace{\mathsf{nn}\ldots\mathsf{n}}_{a-2 \text{ times}}\ \gamma_{2,1}\gamma_{2,2}\ldots\gamma_{2,m})}^{t_{a-1}} \ldots \\ & \overbrace{\mathit{trans}(\mathsf{n}\gamma_{a-1,1}\gamma_{a-1,2}\ldots\gamma_{a-1,m})}^{t_2} \\ & \overbrace{\mathit{trans}(\gamma_{a,1}\gamma_{a,2}\ldots\gamma_{a,m})}^{t_1}. \end{aligned}$$

Trans produces quite complicated terms as the following example shows:

$$\begin{aligned} &\mathit{Trans} \begin{vmatrix} \mathsf{y} & \mathsf{n} & \mathsf{y} \\ \mathsf{n} & \mathsf{y} & \mathsf{y} \\ \mathsf{n} & \mathsf{n} & \mathsf{y} \end{vmatrix} \\ &\quad = \mathit{trans}(\mathsf{nnyny})\, \mathit{trans}(\mathsf{nnyy})\, \mathit{trans}(\mathsf{nny}) \\ &\quad = \mathsf{B}_4\mathsf{S}\big(\mathsf{B}_3\mathsf{C}(\mathsf{B}_2\mathsf{S}(\mathsf{B}_1\mathsf{CK}))\big) \\ &\qquad \big(\mathsf{B}_3\mathsf{S}(\mathsf{B}_2\mathsf{S}(\mathsf{B}_1\mathsf{CK}))\big)\big(\mathsf{B}_2\mathsf{S}(\mathsf{B}_1\mathsf{CK})\big). \end{aligned}$$

Proposition 1. *The function Trans is correct in the sense that if*

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

where γ is a yn*-array and the Q_j, for $1 \leqslant j \leqslant m$, are as specified by the way γ contracts, then*

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

Proof. Let t_i, for $1 \leqslant i \leqslant a$, be as shown above and let $P_i^1 \equiv P_i$, for $1 \leqslant i \leqslant m$. Then

$$t_a \ldots t_1 P_1^1 P_2^1 \ldots P_m^1 P_{m+1} \ldots P_{m+a} \\ \to t_{a-1} \ldots t_1 P_1^2 P_2^2 \ldots P_m^2 P_{m+2} \ldots P_{m+a},$$

where $P_j^2 \equiv P_j^1\ P_{m+1}$, if $\gamma_{1,j} = \mathsf{y}$, and $P_j^2 \equiv P_j^1$, if $\gamma_{1,j} = \mathsf{n}$, because t_a is a yn-string which contracts in this way,

$$\to t_{a-2} \ldots t_1 P_1^3 P_2^3 \ldots P_m^3 P_{m+3} \ldots P_{m+a},$$

where $P_j^3 = P_j^2 P_{m+2}$, if $\gamma_{2,j} = \mathsf{y}$, and $P_j^3 \equiv P_j^2$, if $\gamma_{2,j} = \mathsf{n}$, because t_{a-1} is a yn-string,

$$\to \cdots \\ \to P_1^{a+1} P_2^{a+1} \ldots P_m^{a+1},$$

where $P_j^{a+1} \equiv P_j^a P_{m+a}$, if $\gamma_{a,j} = \mathsf{y}$, and $P_j^{a+1} \equiv P_j^a$, if $\gamma_{a,j} = \mathsf{n}$, because t_1 is a yn-string. Thus,

$$P_j^{a+1} \equiv P_j^1 P_{h_j(1)} P_{h_j(2)} \ldots P_{h_j(r_j)},$$

where $P_{h_j(1)}$ is the $h_j(1)$th term in the list P_{m+1}, $P_{m+2}, \ldots, P_{m+a}$, where $h_j(1)$ is the position of the first occurrence of the letter y in $\gamma_{1,j}\gamma_{2,j} \ldots \gamma_{a,j}$, and $P_{h_j(2)}$ is the $h_j(2)$th term in the same list, where $h_j(2)$ is the position of the second occurrence of the letter y in $\gamma_{1,j}\gamma_{2,j} \ldots \gamma_{a,j}$ and so on. $h_j(r_j)$ is the position of the final occurrence of y in $\gamma_{1,j}\gamma_{2,j} \cdots \gamma_{a,j}$. Thus, r_j is the total number of occurrences of y in $\gamma_{1,j}\gamma_{2,j} \ldots \gamma_{a,j}$. Thus, $r_j = yc(\gamma_{1,j}\gamma_{2,j} \ldots \gamma_{a,j})$ and, for $1 \leqslant k \leqslant r_j$, $h_j(k) = g_j(k)$, where $g_j(k)$ is the function defined in the context of explaining how yn-arrays contract, namely $g_j(k) = a + \mathit{posy}(k, \gamma_{1,j}\gamma_{2,j} \cdots \gamma_{a,j})$. $\square$

5. Abstraction

In order to present an abstraction algorithm that produces abstracts containing yn-arrays two functions have to be defined: $tv([\vec{x}], P)$ returns the total number of variables in the list $\vec{x}$ occurring in P and $\mathit{inx}(i, [\vec{x}], P)$ returns the index of the ith variable in the list $\vec{x}$ occurring in P. For example, $tv([x_1, x_2, x_3], x_1x_3) = 2$ and $\mathit{inx}(1, [x_1, x_2, x_3], x_2x_3(x_1x_2)) = 2$. Algorithm (M) is shown in Fig. 1. Note that a different algorithm would result if it was not a requirement for P_1 to be an atom. 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 n that it does not. An example of (M) should clarify its operation:

$$[x_1, x_2, x_3] x_1 (x_2 x_1)(x_3 x_1) x_2 \\ = \begin{vmatrix} \mathsf{y} & \mathsf{y} & \mathsf{y} & \mathsf{n} \\ \mathsf{n} & \mathsf{y} & \mathsf{n} & \mathsf{y} \\ \mathsf{n} & \mathsf{n} & \mathsf{y} & \mathsf{n} \end{vmatrix} \mathsf{I} \big([x_1, x_2]\ x_2 x_1\big)\big([x_1, x_3]\ x_3 x_1\big)\ \mathsf{I} \\ = \begin{vmatrix} \mathsf{y} & \mathsf{y} & \mathsf{y} & \mathsf{n} \\ \mathsf{n} & \mathsf{y} & \mathsf{n} & \mathsf{y} \\ \mathsf{n} & \mathsf{n} & \mathsf{y} & \mathsf{n} \end{vmatrix} \mathsf{I} \left(\begin{vmatrix} \mathsf{n} & \mathsf{y} \\ \mathsf{y} & \mathsf{n} \end{vmatrix} \mathsf{I}\,\mathsf{I} \right) \left(\begin{vmatrix} \mathsf{n} & \mathsf{y} \\ \mathsf{y} & \mathsf{n} \end{vmatrix} \mathsf{I}\,\mathsf{I} \right) \mathsf{I}.$$

The top row yyyn of the 3×4 yn-array shows the pattern of occurrences of the variable x_1 in the primal components of the input term. Similarly, the second row nyny shows the pattern of occurrences of the variable x_2 in the primal components of the input term and the third row does the same for the variable x_3.

Algorithm (M) has the property that $([\vec{x}]P)\vec{x} \to P$. This is Proposition 2 and the proof is by induction on $\phi(a, rpv([\vec{x}], P))$, where $\phi : \mathbb{N}_1 \times \mathbb{N} \to \mathbb{N}_1$ is a total bijection. ($\mathbb{N}$ is the set of all non-negative whole numbers and $\mathbb{N}_1$ is the set of all positive whole numbers.) The function ϕ is defined thus:

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

Proposition 2. $([\vec{x}]P)\vec{x} \to P$.

Proof. Let $[\vec{x}] = [x_1, x_2, \ldots, x_a]$ and $P \equiv P_1 P_2 \ldots P_m$, where P_1 is an atom. The proof is by induction on $\phi(a, rpv([\vec{x}], P))$.

In the base case $\phi(a, rpv([\vec{x}], P)) = 1$. Thus, $a = 1$ and $rpv([\vec{x}], P) = 0$. We have that $([x_1]\, P) x_1 \equiv \gamma Q_1 Q_2 \ldots Q_m x_1$, where γ and the Q_j, for $1 \leqslant j \leqslant$

In this algorithm P_1 must be an atom.

$[x_1, x_2, \ldots, x_a]P_1P_2 \ldots P_m \equiv \gamma Q_1Q_2 \ldots Q_m$,

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

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

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

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

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

Fig. 1. Algorithm (M).

m, are as specified by (M). As $rpv([\vec{x}], P) = 0$, either $P_j \equiv x_1$ or $x_1 \notin FV(P_j)$, for $1 \leqslant j \leqslant m$. If $P_j \equiv x_1$, then $\gamma_{1,j} = \mathsf{y}$ and $Q_j \equiv \mathsf{I}$. If $x_1 \notin FV(P_j)$, then $\gamma_{1,j} = \mathsf{n}$ and $Q_j \equiv P_j$. Thus, $\gamma Q_1Q_2 \ldots Q_m x_1 \to R_1R_2 \ldots R_m$, as specified by the way yn-arrays contract. If $\gamma_{1,j} = \mathsf{y}$, then $R_j \equiv Q_j\ x_1 \equiv \mathsf{I}x_1 \to x_1$. If $\gamma_{1,j} = \mathsf{n}$, then $R_j \equiv Q_j \equiv P_j$. So, $R_1R_2 \ldots R_m \to P_1P_2 \ldots P_m \equiv P$. This establishes the base case.

In the inductive step $\phi(a, rpv([\vec{x}], P)) > 1$. So,

$$\big([x_1, x_2, \ldots, x_a]P\big)x_1x_2 \ldots x_a \equiv \gamma Q_1Q_2 \ldots Q_m x_1x_2 \ldots x_a,$$

where γ and the Q_j, for $1 \leqslant j \leqslant m$, are as specified by (M),

$$\to R_1R_2 \ldots R_m,$$

where the R_j, for $1 \leqslant j \leqslant m$, are determined by the way yn-arrays contract. First, consider the case when $(\forall i \in 1..a)(\forall j \in 1..m) x_i \notin FV(P_j)$. Then, $\gamma_{i,j} = \mathsf{n}$, $Q_j \equiv P_j$ and $R_j \equiv Q_j$. Thus, $R_j \equiv P_j$. Next, consider the case when $(\exists i \in 1..a)(\exists j \in 1..m)\ x_i \in FV(P_j)$. Then, $\gamma_{i,j} = \mathsf{y}$ and either P_j is a variable or a term. If $P_j \equiv x_i$, for some i, then $Q_j \equiv \mathsf{I}$. If P_j is a term, then $Q_j \equiv [x_{f_j(1)}, \ldots, x_{f_j(q_j)}]P_j$. When $\gamma_{i,j} = \mathsf{y}$, then $R_j \equiv Q_j\ x_{g_j(1)}, \ldots, x_{g_j(s_j)}$, where $f_j(k) = inx(k, \vec{x}, P_j)$ and $g_j(k) = posy(k, \gamma_{1,j}\gamma_{2,j} \ldots \gamma_{a,j})$. Thus, $f_j(k) = g_j(k)$, for all k. Also, $f_j(q_j) \leqslant a$, for all j, and $rpv([x_{f_j(1)}, \ldots, x_{f_j(q_j)}], P_j) < rpv([\vec{x}], P)$. Thus,

$$\phi\big(f_j(q_j), rpv\big([x_{f_j(1)}, \ldots, x_{f_j(q_j)}], P_j\big)\big) < \phi\big(a, rpv([\vec{x}], P)\big).$$

Therefore, $R_j \to P_j$, by the inductive hypothesis. The result follows by induction. □

The proof of Proposition 4 makes use of a lemma. Informally, this states that the length of an abstract produced by (M) is not affected by adding extra variables to the bracket prefix which do not occur in the input term.

Lemma 3. *Let* $[\vec{x}] = [x_1, x_2, \ldots, x_a]$ *and* $[\vec{y}] = [y_1, y_2, \ldots, y_b]$. *Then if* $b \leqslant a$ *and* $(\forall k \in 1..b)(\exists i \in 1..a)(x_i \equiv y_k)$ *and* $(\forall k, l \in 1..b)(\forall i, j \in 1..a)$ (*if* $x_i \equiv y_k$ *and* $x_j \equiv y_l$ *and* $i < j$, *then* $k < l$) *and* $(\forall i \in 1..a)$ (*if* $x_i \in FV(P)$, *then* $(\exists k \in 1..b)(x_i \equiv y_k)$), *then* $\#([\vec{x}]P) = \#([\vec{y}]P)$.

Proof. Let $P \equiv P_1P_2 \ldots P_m$, where P_i is an atom. Then $[\vec{x}]P \equiv \gamma Q_1Q_2 \ldots Q_m$, where γ and the Q_j, for $1 \leqslant j \leqslant m$, are as specified by (M) and $[\vec{y}]P \equiv \delta R_1R_2 \ldots R_m$, where δ and the R_j, for $1 \leqslant j \leqslant m$, are also as specified by (M).

To establish that $(\forall j \in 1..m)\ Q_j \equiv R_j$ we consider two cases. (1) If P_j contains none of the abstraction variables, then $Q_j \equiv P_j$ and $R_j \equiv P_j$, both by (M). (2) When (M) is applied recursively only those variables that actually occur in the primal component P_j are included in the bracket prefix that is passed to the recursive call of the algorithm. Thus, again, $Q_j \equiv R_j$.

The only difference between the abstracts is that γ is an $a \times m$ yn-array, whereas δ is an $b \times m$ yn-array. If $b < a$, then γ has $a - b$ extra rows each of which consists entirely of occurrences of n. These correspond to the extra variables in the bracket prefix $[\vec{x}]$ which, *ex hypothesi*, do not occur in P. As $\#\gamma = \#\delta$, we have that $\#([\vec{x}]P) = \#([\vec{y}]P)$. $\square$

Proposition 4. $\#([\vec{x}]P) = 1 + \#P + rpv([\vec{x}], P)$.

Proof. Let $[\vec{x}] = [x_1, x_2, \ldots, x_a]$ and $P \equiv P_1 P_2 \ldots P_m$, where P_i is an atom. The proof is by induction on $rpv([\vec{x}], P)$.

In the base case $rpv([\vec{x}], P) = 0$. Thus, for $1 \leqslant j \leqslant m$, either $P_j \equiv x_i$, for some i such that $1 \leqslant i \leqslant a$, or $x_i \notin FV(P_j)$, for any i such that $1 \leqslant i \leqslant a$. Also, $[\vec{x}]P \equiv \gamma Q_1 Q_2 \ldots Q_m$, where γ and the Q_j, for $1 \leqslant j \leqslant m$, are as specified by (M). If $P_j \equiv x_i$, then $Q_j \equiv \mathsf{I}$. If $x_i \notin FV(P_j)$, then $Q_j \equiv P_j$. Thus,

$$\#\big([\vec{x}]P\big) = 1 + \sum_{j=1}^{m} \#P_j = 1 + \#P + rpv\big([\vec{x}], P\big).$$

This establishes the base case.

In the inductive step, we have that $[\vec{x}]P \equiv \gamma Q_1 Q_2 \ldots Q_m$, where γ and the Q_j, for $1 \leqslant j \leqslant m$, are as specified by (M). For $1 \leqslant j \leqslant m$, if $P_j \equiv x_i$, for some i such that $1 \leqslant i \leqslant a$, or $x_i \notin FV(P_j)$, for any i such that $1 \leqslant i \leqslant a$, then $\#Q_j \equiv \#P_j$. Thus,

$$\begin{aligned}
&\#\big([\vec{x}]P\big) \\
&\quad = 1 + \sum_{j=1}^{m} \textbf{if } (\exists i \in 1..a)\ x_i \in FV(P_j) \quad \textbf{and} \\
&\qquad\qquad (\forall i \in 1..a)\ x_i \neq P_j \\
&\qquad\qquad \textbf{then } \#\big([x_{f_j(1)}, x_{f_j(2)}, \ldots, x_{f_j(q_j)}]P_j\big) \\
&\qquad\qquad\quad \textbf{else } \#P_j \\
&\quad = 1 + \sum_{j=1}^{m} \textbf{if } (\exists i \in 1..a)\ x_i \in FV(P_j) \quad \textbf{and} \\
&\qquad\qquad (\forall i \in 1..a)\ x_i \neq P_j \\
&\qquad\qquad \textbf{then } \#\big([\vec{x}]P_j\big) \textbf{ else } \#P_j,
\end{aligned}$$

by Lemma 3,

$$\begin{aligned}
&\quad = 1 + \sum_{j=1}^{m} \textbf{if } (\exists i \in 1..a)\ x_i \in FV(P_j) \quad \textbf{and} \\
&\qquad\qquad (\forall i \in 1..a)\ x_i \neq P_j \\
&\qquad\qquad \textbf{then } 1 + \#P_j + rpv\big([\vec{x}], P_j\big) \textbf{ else } \#P_j,
\end{aligned}$$

by the inductive hypothesis,

$$\begin{aligned}
&= 1 + \sum_{j=1}^{m} \#P_j + \sum_{j=1}^{m} \textbf{if } (\exists i \in 1..a)\ x_i \in FV(P_j) \\
&\qquad\qquad \textbf{and } (\forall i \in 1..a)\ x_i \neq P_j \\
&\qquad\qquad \textbf{then } 1 + rpv\big([\vec{x}], P_j\big) \textbf{ else } 0 \\
&= 1 + \#P + rpv\big([\vec{x}], P\big),
\end{aligned}$$

by the property of rpv mentioned above. This establishes the inductive step. The result follows by induction. $\square$

6. Conclusion

The most popular way of judging the efficiency of an abstraction algorithm is by considering the length of the abstract produced. It has been correctly argued, in my opinion, that by itself this is a very crude measure of efficiency [8, pp. 148–159]. It is, therefore, only one of the factors that we need to take into account when comparing algorithms. If (M) is applied to P, the length of the abstract produced is $1 + \#P + rpv([\vec{x}], P)$. This assumes that the length of a yn-array is 1. This is reasonable if a and m are small, as they usually are when the algorithm is used to implement a functional language, but the larger a and m become the more problematic this assumption becomes. The number of yn-arrays in an abstract is $1 + rpv([\vec{x}], P)$ and the largest of these is an $a \times m$ bit array. The maximum value that $rpv([\vec{x}], P)$ can take, if $\#P \geqslant 2$, is $\#P - 2$. Thus, the space required to store these arrays is not greater than $(\#P - 1)(a \times m)$ bits. Joy, Rayward-Smith and Burton [5, Table 1, p. 216] present the lengths of abstracts produced by various algorithms and (M) is comparable to the best of them. The way in which (M) operates is, however, considerably simpler than its rivals.

It should be noted that $1 + rpv([\vec{x}], P)$ is also the number of times that (M) is called. This is relevant when considering the length of time needed to produce the abstract. Similar information for other algorithms is not readily available, but my experience with some of the best known suggests that they are called many more times than this when applied to the same input terms. This is an area where more research needs to be done. It would also be useful to know how (M) performs in practice and how it compares empirically with other abstraction methods and with other ways of implementing a functional language.

Even if it turns out that yn-arrays are not a practicable way of implementing a functional language, they do have a certain theoretical interest and many fascinating and unusual properties, as I am beginning to discover [3].

References

[1] H.B. Curry, R. Feys, in: Combinatory Logic, Vol. 1, North-Holland, Amsterdam, 1958.

[2] A. Diller, Making abstraction behave by representing combinators, Research Report CSR-99-12, School of Computer Science, University of Birmingham, 1999.

[3] A. Diller, Investigations into iconic representations of combinators, in: J. Blanco (Ed.), Argentine Workshop on Theoretical Computer Science (WAIT2000) Proceedings: Tandil, September 4–9, 2000, SADIO, Buenos Aires, 2000, pp. 52–62.

[4] J.R. Hindley, J.P. Seldin, Introduction to Combinators and λ-calculus, Cambridge University Press, Cambridge, 1986.

[5] M.S. Joy, V.J. Rayward-Smith, F.W. Burton, Efficient combinator code, Comput. Languages 10 (1985) 211–224.

[6] J.R. Kennaway, M.R. Sleep, Variable abstraction in $O(n \log n)$ space, Inform. Process. Lett. 24 (1987) 343–349.

[7] D. Stevens, Variable substitution with iconic combinators, in: A.M. Borzyszkowski, S. Sokołowski (Eds.), Mathematical Foundations of Computer Science, in: Lecture Notes in Computer Science, Vol. 711, Springer, Berlin, 1993, pp. 724–733.

[8] D. Stevens, A generalization of Turner's combinator-based technique for implementing a functional language, PhD thesis, School of Computer Science, University of Birmingham, 1996.

[9] D.A. Turner, A new implementation technique for applicative languages, Software—Practice and Experience 9 (1979) 31–49.