Making Abstraction Behave by Rerepresenting Combinators

Antoni Diller

November 5, 1999

Abstract

When bracket abstraction is being used to implement a functional programming language, it is desirable, according to Turner, that the algorithm used produces short abstracts, uses only a finite number of combinators, is uni-variate and also well-behaved under self-composition. In this paper I show that it is impossible to devise an algorithm using a finite number of combinators that is always well-behaved under self-composition. It is better to retain the property of being well-behaved under self-composition, but then we need to choose our infinite set of combinators carefully. By using an iconic representation for combinators an easily implementable set can be devised. The resulting algorithm, namely (**yn**), produces abstracts that are never longer than those produced by Turner's algorithm and it is always well-behaved. By further exploiting the iconic notation an even better algorithm, namely (**yin**), can be devised, but this is no longer so well-behaved.

1 Introduction

Bracket abstraction, denoted by $[x]\ X$, in weak combinatory logic is a syntactic operation which removes a variable x from a term X. Abstraction algorithms can be designed for a variety of purposes and algorithms that are going to be used differently may require distinct properties. Turner [11] is interested in the use of abstraction to implement a functional programming language and so am I. He lays down a number of requirements that such an algorithm has to possess. It should produce short abstracts, make use of only a finite number of combinators and be uni-variate. Although Turner wants a uni-variate algorithm, he requires that this algorithm should be well-behaved under what he calls *self-composition* [11, p. 270]. However, as I show below, it is impossible for an algorithm using only a finite number of combinators to be always well-behaved under self-composition. Before showing this I need first to present Turner's algorithm. (One of the new algorithms to be presented in this paper, namely (**yn**), is always well-behaved in this way, but it uses combinators drawn from a countably infinite set.)

Although a large number of algorithms have been devised since the publication of Turner's article, his is still surprisingly good. Bunder [2], for example, after surveying a number of algorithms, concludes that Turner's is one of the best and still of substantial interest [2, p. 657]. Turner's algorithm, when applied to relatively short terms (Bunder is not more precise), generally produces the shortest abstracts. I have no quarrel with this analysis of Bunder's and will not reproduce his discussion here. Both of the algorithms to be presented in this paper never produce abstracts longer than those produced by Turner's algorithm. (Joy, Rayward-Smith and Burton [6] work out the efficiency of a large number of algorithms including Turner's.)

Turner begins his presentation of his algorithm of choice by considering the following algorithm [11, p. 268], which elsewhere [4, pp. 93–94] I have called *algorithm (A)*:

$$\begin{aligned} [x]\ x &= \mathsf{I}, \\ [x]\ e &= \mathsf{K}\ e, \quad \text{if } e \text{ is an atom distinct from } x, \\ [x]\ P\ Q &= \mathsf{S}\ ([x]\ P)\ ([x]\ Q). \end{aligned}$$

This algorithm 'tends to produce needlessly long-winded abstracts' [11, p. 268], but the length of the abstract that it produces can be considerably shortened by the use of the following group of optimisations [11, p. 268]:

$$\begin{aligned} \mathsf{S}\ (\mathsf{K}\ P)\ (\mathsf{K}\ Q) &\Rightarrow \mathsf{K}\ (P\ Q), \\ \mathsf{S}\ (\mathsf{K}\ P)\ \mathsf{I} &\Rightarrow P, \\ \mathsf{S}\ (\mathsf{K}\ P)\ Q &\Rightarrow \mathsf{B}\ P\ Q, \\ \mathsf{S}\ P\ (\mathsf{K}\ Q) &\Rightarrow \mathsf{C}\ P\ Q. \end{aligned}$$

Turner calls this second algorithm, comprising of algorithm (A) and his first group of optimisations, *the Curry algorithm* [11, p. 269]. The Curry algorithm is sometimes

presented in the following form:

$$\begin{aligned}
[x]\ E &= \mathsf{K}\ E,\\
[x]\ x &= \mathsf{I},\\
[x]\ E\ x &= E,\\
[x]\ E\ X &= \mathsf{B}\ E\ ([x]\ X),\\
[x]\ X\ E &= \mathsf{C}\ ([x]\ X)\ E,\\
[x]\ X\ Y &= \mathsf{S}\ ([x]\ X)\ ([x]\ Y),
\end{aligned}$$

where E is a term in which x does not occur and X and Y are terms in which x definitely does occur. This is Curry's algorithm (abcdef) [3, p. 190] and elsewhere [4, p. 96] I call it *algorithm (B)*. The Curry algorithm is considerably better than algorithm (A), but it can still produce large abstracts. To shorten the length of the abstract Turner introduced a second group of optimisations:

$$\begin{aligned}
\mathsf{S}\ (\mathsf{B}\ K\ P)\ Q &\Rightarrow \mathsf{S}'\ K\ P\ Q,\\
\mathsf{B}\ (K\ P)\ Q &\Rightarrow \mathsf{B}'\ K\ P\ Q,\\
\mathsf{C}\ (\mathsf{B}\ K\ P)\ Q &\Rightarrow \mathsf{C}'\ K\ P\ Q,
\end{aligned}$$

where K is a term consisting entirely of constants [11, p. 270]. Turner's algorithm is sometimes presented in the following way [7, p. 2]:

$$\begin{aligned}
[x]\ E &= \mathsf{K}\ E,\\
[x]\ x &= \mathsf{I},\\
[x]\ E\ x &= E,\\
[x]\ E\ F\ X &= \mathsf{B}'\ E\ F\ ([x]\ X),\\
[x]\ E\ X\ F &= \mathsf{C}'\ E\ ([x]\ X)\ F,\\
[x]\ E\ X\ Y &= \mathsf{S}'\ E\ ([x]\ X)\ ([x]\ Y),\\
[x]\ E\ X &= \mathsf{B}\ E\ ([x]\ X),\\
[x]\ X\ E &= \mathsf{C}\ ([x]\ X)\ E,\\
[x]\ X\ Y &= \mathsf{S}\ ([x]\ X)\ ([x]\ Y),
\end{aligned}$$

where x does not occur in either E or F, but it does occur in X and Y. Elsewhere [4, p. 98] I have called this *algorithm (C)*. Before Bunder's article [2] it was widely assumed that algorithm (B) is equivalent to algorithm (A) together with Turner's first group of optimisations and that algorithm (C) is equivalent to algorithm (A) together with both of Turner's two groups of optimisations. This, however, is not correct. For example, using algorithm (B) to abstract x from $\mathsf{S}(\mathsf{K}u)v$ produces $\mathsf{K}(\mathsf{S}(\mathsf{K}u)v)$, whereas algorithm (A) followed by Turner's first group of optimisations produces $\mathsf{K}(\mathsf{B}uv)$. Furthermore, using algorithm (C), the result of abstracting x from $\mathsf{B}(\mathsf{B}v)w$ is $\mathsf{K}(\mathsf{B}(\mathsf{B}v)w)$, whereas algorithm (A) followed by Turner's two groups of optimisations produces $\mathsf{K}(\mathsf{B}'\mathsf{B}vw)$. (Note that these counter-examples are different from those that Bunder gives.)

The Curry algorithm is considerably better than algorithm (A), yet it is not well-behaved under what Turner calls *self-composition* [11, p. 270]. What he means by this can be illustrated by means of the following example [11, p. 269]. Consider a term of the form P_1P_2 from which we want to abstract the variables $x_1, x_2, \ldots, x_a$. We assume that each of these variables occurs at least once in both P_1 and P_2. Let P_i' be $[x_1]P_i$, P_i'' be $[x_2]([x_1]P_i)$, and so on. Using the Curry algorithm we get the following sequence of abstracts, where the length of the abstract is at least proportional to the square of the number of variables abstracted [11, p. 269]:

P_1P_2	initial term
$\mathsf{S}P_1'P_2'$	first abstract
$\mathsf{S}(\mathsf{BS}P_1'')P_2''$	second abstract
$\mathsf{S}(\mathsf{BS}(\mathsf{B}(\mathsf{BS})P_1'''))P_2'''$	third abstract
$\mathsf{S}(\mathsf{BS}(\mathsf{B}(\mathsf{BS})(\mathsf{B}(\mathsf{B}(\mathsf{BS}))P_1''''))P_2''''$	fourth abstract

Using Turner's algorithm we get the following sequence, where the lengths of the abstracts form a linear progression [11, p. 269]:

P_1P_2	initial term
$\mathsf{S}P_1'P_2'$	first abstract
$\mathsf{S'S}P_1''P_2''$	second abstract
$\mathsf{S'}(\mathsf{S'S})P_1'''P_2'''$	third abstract
$\mathsf{S'}(\mathsf{S'}(\mathsf{S'S}))P_1''''P_2''''$	fourth abstract

He notes that by using his algorithm the structure of the abstract is left unaltered in the form KQ_1Q_2, where K is a term consisting entirely of combinators [11, p. 269]. (Turner wants a uni-variate algorithm that is well-behaved under self-composition rather than a multi-variate one because 'in the compilation process the need to abstract on different variables arises at successive stages' rather than all at the same time [11, p. 270].) Although this is true for terms of the form P_1P_2, it is not the whole story. In fact, it is impossible to devise an algorithm which uses only a finite number of combinators and which is always well-behaved under self-composition. If we consider a term of the form $P_1P_2P_3$, then we see that Turner's algorithm no longer possesses this property.

$P_1P_2P_3$	initial term
$\mathsf{S}(\mathsf{S}P_1'P_2')P_3'$	first abstract
$\mathsf{S'S}(\mathsf{S'S}P_1''P_2'')P_3''$	second abstract
$\mathsf{S'}(\mathsf{S'S})(\mathsf{S'}(\mathsf{S'S})P_1'''P_2''')P_3'''$	third abstract
$\mathsf{S'}(\mathsf{S'}(\mathsf{S'S}))(\mathsf{S'}(\mathsf{S'}(\mathsf{S'S}))P_1''''P_2'''')P_3''''$	fourth abstract

The abstracts here do not have the form $KQ_1Q_2Q_3$ and so are not well-behaved under self-composition. (A pattern can be discerned, namely each abstract has the form $K_1(K_2Q_1Q_2)Q_3$, though this is not so useful as having the form $KQ_1Q_2Q_3$.) To produce abstracts with this form we would need to have combinators S_2 and Φ_3 at our disposal:

$$\mathsf{S}_2QRST \to QT(RT)(ST),$$
$$\Phi_3PQRST \to P(QT)(RT)(ST).$$

(These are defined by Curry and Feys [3, p. 169]. Note that $\mathsf{S}_1 = \mathsf{S}$ and that Turner's S' is Φ or, equivalently, Φ_2.) The clauses in an abstraction algorithm corresponding to these combinators are the following:

$$[x]XYZ \to \mathsf{S}_2([x]X)([x]Y)([x]Z),$$
$$[x]EXYZ \to \Phi_3 E([x]X)([x]Y)([x]Z),$$

where x does not occur in E, but does in X, Y and Z. An algorithm augmented with these clauses would produce the following abstracts:

$P_1P_2P_3$	initial term
$\mathsf{S}_2P_1'P_2'P_3'$	first abstract
$\Phi_3\mathsf{S}_2P_1''P_2''P_3''$	second abstract
$\Phi_3(\Phi_3\mathsf{S}_2)P_1'''P_2'''P_3'''$	third abstract
$\Phi_3(\Phi_3(\Phi_3\mathsf{S}_2))P_1''''P_2''''P_3''''$	fourth abstract

These abstracts do have the form $KQ_1Q_2Q_3$, but if we consider a term of the form $P_1P_2P_3P_4$ then the following abstracts are produced:

$P_1P_2P_3P_4$	initial term
$\mathsf{S}_2(\mathsf{S}P_1'P_2')P_3'P_4'$	first abstract
$\Phi_3\mathsf{S}_2(\mathsf{S}'\mathsf{S}P_1''P_2'')P_3''P_4''$	second abstract
$\Phi_3(\Phi_3\mathsf{S}_2)(\mathsf{S}'(\mathsf{S}'\mathsf{S})P_1'''P_2''')P_3'''P_4'''$	third abstract
$\Phi_3(\Phi_3(\Phi_3\mathsf{S}_2))(\mathsf{S}'(\mathsf{S}'(\mathsf{S}'\mathsf{S}))P_1''''P_2'''')P_3''''P_4''''$	fourth abstract

Clearly, an algorithm that employs only a finite number of combinators cannot always be well-behaved under self-composition. Algorithm (**yn**), to be presented below, is always well-behaved in this way. This is achieved by using countably many combinators. Using an idea of Stevens's [10], these are represented iconically. This has two main advantages, namely their behaviour can be read off from their representation and they can be easily implemented.

Bracket abstraction usually satisfies the following property, where $\to$ represents weak reduction:

$$([x]X)x \to X. \tag{1}$$

Turner's way of presenting abstraction algorithms has them produce abstracts which do not always satisfy the property (1). They do, however, satisfy the following weaker property:

$$([x]X)x =_{\beta\eta} X. \tag{2}$$

Bunder follows Turner in producing several algorithms all of which satisfy (2) rather than (1). He devises algorithms better than Turner's by adding extra optimisations. Both of the algorithms presented in this paper satisfy property (1).

To summarise what I do in this paper: Two uni-variate algorithms are presented. Both of them produce abstracts that are never longer than those produced by Turner's algorithm. Algorithm (**yn**) is well-behaved under self-composition. Both of the algorithms make use of combinators drawn from a fixed set of countably many combinators. These are, however, well suited to computer implementation being, in the case of (**yn**), easily represented as bit strings. Both of them satisfy property (1).

2 Fixing Terminology

A *term of combinatory logic* (or *CL-term* or *term*) is defined recursively as follows:

(a) Every variable is a CL-term.

(b) Every constant is a CL-term.

(c) If P and Q are CL-terms, then so is $(P\ Q)$.

In this paper the letters E, F, K, M, N, P, Q, R, S, T, X, Y and Z, sometimes with subscripts or primes, will be used for arbitrary terms. An *atom* is either a variable or a constant and a term of the form (PQ) is known either as a *combination* or an *application*. When writing terms that are combinations, the outermost pair of parentheses will usually be omitted. Furthermore, application associates to the left, so it is possible to write $PQRST$ instead of $(((PQ)R)S)T$. The idea of a subterm is defined as follows: P is a subterm of P and P is a subterm of QR if either P is a subterm of Q or P is a subterm of R.

In order to complete the definition of a term it is only necessary to specify what constitutes a variable and what constitutes a constant. In this paper the letters u, v, w, x, y and z, with or without subscripts, will be used as variables. The notation used for constants later on in this paper is unusual and will be explained in due course, but to begin with combinatory logic will be presented in the conventional way. In this combinators are referred to by means of identifiers which consist of a single letter. These are usually printed using a bold, sans serif font. There are several versions of combinatory logic and one of the ways in which they differ involves the set of constants that are taken to be primitive or basic. In one common version, and this is the version that will be presented here, the only two primitive constants are the letters **K** and **S**. These stand for combinators. Because combinatory logic contains no variable-binding operators every variable that occurs in a term is *free*. The notation $FV(P)$ is used to denote the set of free variables in the term P. The *length* of a term is the number of occurrences of atoms that it contains. If P is a term, then the length of P is denoted by $\#P$ and is defined like this:

$$\begin{aligned} \#(P) &= 1, && \text{if } P \text{ is a variable or a constant;} \\ \#(PQ) &= \#P + \#Q, && \text{otherwise.} \end{aligned}$$

Every term P can be uniquely expressed in the form $P_1\ P_2\ \ldots\ P_m$, where P_1 is an atom and $m \geq 1$. Note that it is not just combinations that can be expressed in this form. When P is an atom, then $m = 1$ and P_1 is the same as P. When a term is expressed as $P_1\ P_2\ \ldots\ P_m$, where P_1 is an atom, then P_1 is known as the *head* or *leading element* of P and, following Abdali [1, p. 223], the terms P_i, for $1 \leq i \leq m$, are called the *primal components* of P. Note that the head of P is also one of its primal components. For example, the primal components of the term $x\ y\ (z\ y)\ (w\ x\ y)$ are the terms x, y, $z\ y$ and $w\ x\ y$.

Let P be a term which, if $\#P > 1$, is represented as a binary tree whose internal nodes are application nodes. Then $rp(P)$ is the number of application nodes in P

whose right child is also an application node. If P is an atom, then $rp(P)$ is taken to be zero. If P is represented in a linear form using the fewest possible parentheses, then $rp(P)$ is equal to the number of right parentheses in P. (Equivalently, $rp(P)$ is the number of left parentheses in P or half the total number of parentheses in P.) For example, $rp(w\ x\ y\ z) = 0$ and $rp(x\ (x\ (x\ x))) = 2$. Note that in these examples the left parenthesis immediately following the *rp* operator and its matching right parenthesis are used solely in order to indicate the argument of the *rp* operator and are not included in the *rp* count. If $\#P \geq 2$, then the maximum value that $rp(P)$ can take is $\#P - 2$ (because the number of application nodes in a tree representation of P is $\#P - 1$) and the minimum value that $rp(P)$ can take is 0. We also have that if $P = P_1\ P_2\ \ldots\ P_m$, where P_1 is an atom, then

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

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 term whose main operator is the application corresponding to that node. Then $rpx(x, P)$ is the number of application nodes in P whose right child is an application node which also covers x. If P is represented in a linear form using the fewest possible parentheses, then $rpx(x, P)$ is equal to half the number of parentheses that enclose subterms containing the variable x. For example, $rpx(x, y(uv(wy))) = 0$ and $rpx(x, y(x(vy)y)x) = 1$. If $\#P \geq 2$, then the maximum value of $rpx(x, P)$ is $\#P - 2$ and its minimum value is 0. Let $P = P_1\ P_2\ \ldots\ P_m$, where P_1 is an atom. Then

$$rpx(x, P) = \sum_{i=1}^{m} \textbf{if } P_i \neq x \textbf{ and } x \in FV(P_i) \textbf{ then } 1 + rpx(x, P_i) \textbf{ else } 0.$$

3 An Algorithm Using y and n

3.1 Rerepresenting Combinators

Many bracket abstraction algorithms have been proposed in the literature. Although some of these contain interesting ideas, almost all of them use the conventional notation of representing combinators. One exception is Stevens [10]. The method employed here is a variant of Stevens's approach. Stevens introduces a new notation for representing combinators. In one version of this notation the names of combinators are identifiers consisting of two occurrences of one or more of the four letters **o**, **k**, **i** or **s**. He calls such identifiers *iconic representations* [10, p. 727]. Examples of such representations are **ks**, **ii** and **is**. The traditional names of the combinators referred to by these identifiers are **B**, **M** and **O**, respectively. The meaning of an iconic representation depends upon the choice of a particular reference expression. A *reference expression*, for a clause in a bracket abstraction algorithm, is the shape or form of the term to which that clause is applied. Stevens restricts his attention to "square" expressions of various levels. In

this paper I present a variant of Stevens's method which does not use a fixed reference expression. The shape of the reference expression is determined by the form of the term to which the abstraction algorithms presented here are applied and the notation used to refer to combinators is intimately related to the shape of the reference expression that is used.

Instead of using single-letter identifiers for combinators multi-letter identifiers are used. These are strings made up out of the letters **y** and **n** and are known as *yes-no representations*. (The reason why they are called this is explained near the beginning of section 3.2 in which algorithm (**yn**) is discussed.) The collection of all yes-no representations is defined by means of the following left-linear grammar in which A, the start symbol, is the only non-terminal and **y** and **n** are the two terminals:

$$A ::= \mathbf{y} \mid \mathbf{n} \mid A\mathbf{y} \mid A\mathbf{n}.$$

In the case of the standard representations for combinators, such as **K**, **I**, **B** and so on, the reduction property of a combinator has to be explicitly given, whereas in the case of the yes-no representations the reduction property of a combinator can be read off from its representation, thus:

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

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

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

As an example, consider the yes-no representation **ynyyn**:

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

The *size* of a yes-no representation β is the number of occurrences the letters **y** or **n** that it contains and it is denoted by $size(\beta)$. For example, $size(\mathbf{ynyyn}) = 5$. The arity of any yes-no representation β is $1 + size(\beta)$. The Greek letters α, β and γ will be used for iconic representations. If α is an iconic representation, then α_i, for $1 \leq i \leq size(\alpha)$, is the ith iconic letter in α.

It is possible to translate every yes-no representation into a combination of the constants **K** and **S**. First, the constants **B**, **I** and **C** are defined in terms of **K** and **S**. Then we define the series of constants $\mathbf{B}_i$, for $i \geq 1$. These are taken from a book by Smullyan [9, pp. 316–317 and 341] and they can be defined as follows:

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

The translation proceeds as follows:

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

where $i = size(\alpha)$. For example, $trans(\mathbf{ynyyn}) = \mathbf{B}_4\ \mathbf{C}\ (\mathbf{B}_3\ \mathbf{S}\ (\mathbf{B}_2\ \mathbf{S}\ (\mathbf{B}_1\ \mathbf{C}\ (\mathbf{B}\ \mathbf{I}))))$. The new abstraction algorithm presented in this section makes use of yes-no representations in order to refer to combinators. In discussing this algorithm the collection of constants used in the definition of a term will consist of all the yes-no representations and the constant $\mathbf{I}$. Although $\mathbf{I}$ can be defined in terms of yes-no representations, it is more convenient to take it as a primitive constant. Note that as every yes-no representation β is taken to be a constant this means that $\#\beta = 1$. This is acceptable from a mathematical point of view, but from a computing perspective we would need to take account of the amount of space required to store a yes-no representation.

There is some superficial resemblance between yes-no representations and the director strings of Kennaway and Sleep [8]. There are, however, a number of important differences. Chief amongst these is the fact 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 the theory of combinatory logic, but to use a different notation for representing combinators.

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

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

where the Q_i, for $1 \leq i \leq m$, are as given above, then

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

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

$$trans(\beta_1\beta_2\ldots\beta_m)\ P_1\ P_2\ \ldots\ P_m\ P_{m+1} \to Q_1\ Q_2\ \ldots\ Q_m,$$

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

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

The proof is by induction on the size of $\beta_1\beta_2\ldots\beta_m$. In the base case $m = 1$. There are two cases to consider. In the first $\beta_1 = \mathbf{y}$ and in the second $\beta_1 = \mathbf{n}$. When $\beta_1 = \mathbf{y}$, we have that $trans(\mathbf{y})\ P_1\ P_2 = \mathbf{B}\ \mathbf{I}\ P_1\ P_2 \to \mathbf{I}\ (P_1\ P_2) \to P_1\ P_2$. When $\beta_1 = \mathbf{n}$, we have that $trans(\mathbf{n})\ P_1\ P_2 = \mathbf{K}\ P_1\ P_2 \to P_1$. 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 when $m = n - 1$. There are two cases to consider. In the first we look at $\beta_1\beta_2\ldots\beta_{n-1}\mathbf{y}$ and in the second we consider $\beta_1\beta_2\ldots\beta_{n-1}\mathbf{n}$. In the first case we have:

$$\begin{aligned}
&trans(\beta_1\beta_2\ldots\beta_{n-1}\mathbf{y})\ P_1\ P_2\ \ldots\ P_n\ P_{n+1} \\
&= \mathbf{B}_{n-1}\ \mathbf{S}\ trans(\beta_1\beta_2\ldots\beta_{n-1})\ P_1\ P_2\ \ldots\ P_n\ P_{n+1} \\
&\to \mathbf{S}\ (trans(\beta_1\beta_2\ldots\beta_{n-1})\ P_1\ P_2\ \ldots\ P_{n-1})\ P_n\ P_{n+1} \\
&\to trans(\beta_1\beta_2\ldots\beta_{n-1})\ P_1\ P_2\ \ldots\ P_{n-1}\ P_{n+1}\ (P_n\ P_{n+1}) \\
&\to Q_1\ Q_2\ \ldots\ Q_{n-1}\ (P_n\ P_{n+1}),
\end{aligned}$$

where, for $1 \leq i \leq n-1$,

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

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

$$\begin{aligned} & \mathit{trans}(\beta_1\beta_2 \ldots \beta_{n-1}\mathbf{n})\ P_1\ P_2\ \ldots\ P_n\ P_{n+1} \\ = {} & \mathbf{B}_{n-1}\ \mathbf{C}\ \mathit{trans}(\beta_1\beta_2 \ldots \beta_{n-1})\ P_1\ P_2\ \ldots\ P_n\ P_{n+1} \\ \to {} & \mathbf{C}\ (\mathit{trans}(\beta_1\beta_2 \ldots \beta_{n-1})\ P_1\ P_2\ \ldots\ P_{n-1})\ P_n\ P_{n+1} \\ \to {} & \mathit{trans}(\beta_1\beta_2 \ldots \beta_{n-1})\ P_1\ P_2\ \ldots\ P_{n-1}\ P_{n+1}\ P_n \\ \to {} & Q_1\ Q_2\ \ldots\ Q_{n-1}\ P_n, \end{aligned}$$

where, for $1 \leq i \leq n-1$,

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

by the inductive hypothesis. Combining these two cases yields:

$$\mathit{trans}(\beta_1\beta_2 \ldots \beta_n)\ P_1\ P_2\ \ldots\ P_n\ P_{n+1} \to Q_1\ Q_2\ \ldots\ Q_n,$$

where, for $1 \leq i \leq n$,

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

This accords with the reduction rule for $\beta_1\beta_2 \ldots \beta_n$ given above. The result follows by induction. QED.

3.2 Algorithm (yn)

Algorithm (**yn**) is shown in Figure 1. Note that a different algorithm would result if P_1 was not required to be an atom. The reference expressions for the three clauses of algorithm (**yn**) are x, $E\ x$ and $P_1\ P_2\ \ldots\ P_m$, where $x \notin FV(E)$, P_1 is an atom and there is no restriction on whether or not x occurs in any of the P_i, for $1 \leq i \leq m$. An example of the application of algorithm (**yn**) should make its operation clear.

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

In the context of abstraction, when algorithm (**yn**) is being used, the letter **y** means that the primal component to which it corresponds contains the abstraction variable. (The letter β_i, for $1 \leq i \leq m$, is said to *correspond* to the primal component P_i.) That primal component could either be identical to the abstraction variable or it could be a combination that contains the abstraction variable. The letter **n** means that the

In this algorithm E is any term such that $x \notin FV(E)$ and P_1 is an atom. The clauses of this algorithm have to be applied in the order in which they occur here.

$$[x]\ x = \mathbf{I},$$
$$[x]\ E\ x = E,$$
$$[x]\ P_1\ P_2\ \ldots\ P_m = \beta_1\beta_2\ldots\beta_m\ Q_1\ Q_2\ \ldots\ Q_m,$$

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

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

Figure 1: Algorithm (**yn**).

primal component to which it corresponds does not contain the abstraction variable. That primal component could either be an atom distinct from the abstraction variable or it could be a combination that does not contain the abstraction variable. It should be noted that in the context of abstraction the meanings of the letters **y** and **n** are not absolute. They are, rather, relative to a specific algorithm. These comments, however, apply only to the process of abstraction. When it comes to reduction, no matter which algorithm has been used to obtain the abstract, then the meaning of the letters **y** and **n** in the reduction process is always the same. (The reason why yes-no representations are so-called is because the first algorithm involving them that I devised was similar to algorithm (**yn**). In using algorithm (**yn**) to abstract x, say, from a term $P_1\ P_2\ \ldots\ P_m$ one looks at each primal component P_i in turn and asks, 'Does x occur in P_i?' If the answer is 'yes', then the letter β_i is **y**, that is to say, the first letter of the word 'yes', and if the answer is 'no', the letter β_i is **n**, that is to say, the first letter of the word 'no'. It was only after a period of experimentation with different algorithms that I realised that the meaning of the letters **y** and **n** was fixed only in reduction, but variable in abstraction. I decided, however, to keep the name and the form of representation.) Some of the basic properties of algorithm (**yn**) are stated in the following proposition.

Proposition 2 *Let P be any term. Then algorithm (**yn**) has the following properties:*

*(a) If algorithm (**yn**) is applied to P, then it will terminate.*

*(b) Algorithm (**yn**) has the property that $([x]\ P)\ x \to P$.*

*(c) Let algorithm (**yn**\c) be the same as (**yn**) except that the clause $[x]\ E\ x = E$ has been deleted. Then, if $\#P \geq 2$, we have that $\#([x]P) = 1 + \#P + rpx(x, P)$.*

*(d) Using algorithm (**yn**) we have that $\#([x]P) \leq 1 + \#P + rpx(x, P)$.*

*(e) Let $[x]_{\mathrm{C}}\ P$ represent the result of applying algorithm (C) to P and let $[x]_{\mathbf{yn}}\ P$ represent the result of applying algorithm (**yn**) to P. Then $\#([x]_{\mathbf{yn}}\ P) \leq \#([x]_{\mathrm{C}}\ P)$.*

*(f) Using algorithm (**yn**\c) we have that, if $\#P \geq 2$, 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 proofs of parts (a), (b) and (c) are all by induction on the value of $rpx(x, P)$. Part (d) is a straightforward corollary of part (c). The proof of part (e) is by induction on the length of P and proceeds by analysing each clause of algorithm (C) in turn. The proof of part (f) is by induction on a and it makes use of the fact that $rpx(y, [x]P) = rpx(y, P)$.

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 induction on the value of $rpx(x, P)$. In the base case $rpx(x, P) = 0$. Let $P = P_1\ P_2\ \ldots\ P_m$, where $m \geq 2$. For $1 \leq i \leq m$, either $P_i = x$ or P_i is a term such that $x \notin FV(P_i)$. Using algorithm (**yn**\c) we have that $[x]P = \beta\ Q_1\ Q_2\ \ldots\ Q_m$, where the Q_i, for $1 \leq i \leq m$, are as specified in Figure 1. When $P_i = x$, then $Q_i = \mathbf{I}$. When P_i is a term such that $x \notin FV(P_i)$, then $Q_i = P_i$. 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 $P = P_1\ P_2\ \ldots\ P_m$, where $m \geq 2$. For $1 \leq i \leq m$, either $P_i = x$ or P_i is a term such that $x \notin FV(P_i)$ or $P_i \neq x$ and $x \in FV(P_i)$. Using algorithm (**yn**\c) we have that $[x]P = \beta\ Q_1\ Q_2\ \ldots\ Q_m$, where the Q_i, for $1 \leq i \leq m$, are as specified in Figure 1. When $P_i = x$, then $Q_i = \mathbf{I}$. When P_i is a term such that $x \notin FV(P_i)$, then $Q_i = P_i$. When $P_i \neq x$ 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. Note that in this case $\#P_i \geq 2$. We thus have:

$$\begin{aligned}
\#([x]P) &= 1 + \sum_{i=1}^{m} \textbf{if } x = P_i \textbf{ or } x \notin FV(P_i) \textbf{ then } \#P_i \textbf{ else} \\
&\qquad\qquad \textbf{if } x \neq P_i \textbf{ and } x \in FV(P_i) \textbf{ then } 1 + \#P_i + rpx(x, P_i) \\
&= 1 + \#P + \sum_{i=1}^{m} \textbf{if } x \neq P_i \textbf{ and } x \in FV(P_i) \textbf{ then } 1 + rpx(x, P_i) \textbf{ else } 0 \\
&= 1 + \#P + rpx(x, P),
\end{aligned}$$

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

Note that, if $\#P \geq 2$, then the length of $[x]\ P$ can equal $1 + \#P + rp(P)$. This happens, for example, when every atom in P is the same as the abstraction variable and $rpx(x, P)$ has its maximum value. For example, let $Q = x\ (x\ (x\ x))$. Then $[x]\ Q = \mathbf{yy\ I}\ (\mathbf{yy\ I}\ (\mathbf{yy\ I}\ (\mathbf{yy\ I\ I})))$ and $\#([x]\ Q) = 9 = 1 + \#Q + rp(Q)$. Note also that algorithm (**yn**) is well-behaved under self-composition:

$P_1\ P_2\ \ldots\ P_m$ initial term

$\underbrace{\mathsf{yy}\cdots\mathsf{y}}_{m\text{ times}}\ P'_1\ P'_2\ \ldots\ P'_m$ first abstract

$\mathsf{n}\underbrace{\mathsf{yy}\cdots\mathsf{y}}_{m\text{ times}}\ \underbrace{\mathsf{yy}\cdots\mathsf{y}}_{m\text{ times}}\ P''_1\ P''_2\ \ldots\ P''_m$ second abstract

$\mathsf{nn}\underbrace{\mathsf{yy}\cdots\mathsf{y}}_{m\text{ times}}\ \mathsf{n}\underbrace{\mathsf{yy}\cdots\mathsf{y}}_{m\text{ times}}\ \underbrace{\mathsf{yy}\cdots\mathsf{y}}_{m\text{ times}}\ P'''_1\ P'''_2\ \ldots\ P'''_m$ third abstract

$\mathsf{nnn}\underbrace{\mathsf{yy}\cdots\mathsf{y}}_{m\text{ times}}\ \mathsf{nn}\underbrace{\mathsf{yy}\cdots\mathsf{y}}_{m\text{ times}}\ \mathsf{n}\underbrace{\mathsf{yy}\cdots\mathsf{y}}_{m\text{ times}}\ \underbrace{\mathsf{yy}\cdots\mathsf{y}}_{m\text{ times}}\ P''''_1\ P''''_2\ \ldots\ P''''_m$ fourth abstract

This means that not only does algorithm (**yn**) produce abstracts that are never longer than those produced by algorithm (C) but the form of the abstracts that it produces are well suited to repeated abstraction. For example, let $P = x\ y\ z\ x\ x$. Then using algorithm (**yn**) we have:

$$[x]P = \mathsf{ynnyy\ I}\ y\ z\ \mathsf{I\ I};$$
$$[y]([x]P) = \mathsf{nnynnn\ ynnyy\ I\ I}\ z\ \mathsf{I\ I};$$
$$[z]([y]([x]P)) = \mathsf{nnnnynn\ nnynnn\ ynnyy\ I\ I\ I\ I\ I}.$$

Using algorithm (C), however, produces the following sequence of abstracts:

$$[x]P = \mathsf{S(S(C(CI}y)z)\mathsf{I)I};$$
$$[y]([x]P) = \mathsf{C'S(C'S(C'C(CI}z)\mathsf{I)I};$$
$$[z]([y]([x]P)) = \mathsf{C'(C'S)(C'(C'S)(C'C(CI))I)I}.$$

4 An Algorithm Using y, i and n

4.1 Preliminaries

Consider the following example of the application of algorithm (**yn**):

$$[x]\ x\ y\ z\ x\ x = \mathsf{ynnyy\ I}\ y\ z\ \mathsf{I\ I}.$$

To each occurrence of the abstraction variable in the input term there is an occurrence of the combinator **I** in the abstract. The abstract could be shortened if the information conveyed by the presence of **I** could somehow be incorporated in the representation used for the combinator produced by the abstraction algorithm. The algorithm presented in this section uses **yin** combinators that allow this to be done. This algorithm produces the following result:

$$[x]\ x\ y\ z\ x\ x = \mathsf{innii}\ y\ z.$$

The occurrence of the letter **i** in the representation means that the corresponding primal component in the input term is the same as the abstraction variable.

The collection of **yin** representations consists of all strings made up out of the letters **y**, **i** and **n**. It can be defined more formally by the following left-linear grammar, where

the start symbol is B and **y**, **i** and **n** are the three terminals of the language being defined:

$$B ::= \mathbf{y} \mid \mathbf{i} \mid \mathbf{n} \mid B\mathbf{y} \mid B\mathbf{i} \mid B\mathbf{n}.$$

The function $ic(\beta)$ returns the number of occurrences of the letter **i** in the representation β. If β_i, for $1 \leq i \leq m$, is one of the letters **y**, **i** and **n** and $\beta = \beta_1\beta_2\ldots\beta_m$, then $ic(\beta) = \sum_{i=1}^{m}$ **if** $\beta_i = \mathbf{i}$ **then** 1 **else** 0. There are a number of other functions that will be used in what follows.

$ac(x, P)$	This is the number of primal components in P that are equal to x.
$tac(x, P)$	This is the total number of occurrences of x in P.
$posi(j, \beta)$	This is the position of the jth occurrence of the letter **i** in β. If there are fewer than j occurrences of **i** in β, then this function is undefined.

The new iconic representations behave as follows:

$$\beta_1\beta_2\ldots\beta_m\ P_1\ P_2\ \ldots\ P_n \to Q_1\ Q_2\ \ldots\ Q_m,$$

where $n = m + 1 - ic(\beta_1\beta_2\ldots\beta_m)$ and, for $1 \leq i \leq m$, we have that

$$Q_i = \begin{cases} P_n, & \text{if } \beta_i = \mathbf{i}; \\ P_{f(i)}\ P_n, & \text{if } \beta_i = \mathbf{y}; \\ P_{f(i)}, & \text{if } \beta_i = \mathbf{n}; \end{cases}$$

where $f(i) = i - ic(\beta_1\beta_2\ldots\beta_i)$. For example, **innii** $y\ z\ x \to x\ y\ z\ x\ x$.

The iconic representations made up out of **y**, **i** and **n** can be translated into the usual combinators. If the representation β does not contain **i**, then we can use the translation given earlier. If β does contain **i**, then the following translation suffices:

$$trans(\beta_1\beta_2\ldots\beta_m) = \underbrace{\mathbf{B}_2\ \mathbf{B}_2\ \ldots\ \mathbf{B}_2}_{q \text{ times}}\ \mathbf{C}_{a_1}\ \mathbf{C}_{a_2}\ \ldots\ \mathbf{C}_{a_p}\ \gamma_1\gamma_2\ldots\gamma_m\ \underbrace{\mathbf{I}\ \mathbf{I}\ \ldots\ \mathbf{I}}_{p \text{ times}},$$

where

$$\begin{aligned}
p &= ic(\beta_1\beta_2\ldots\beta_m); \\
q &= \sum_{i=1}^{p-1} i = \frac{p(p-1)}{2}; \\
\mathbf{C}_i &= \begin{cases} \mathbf{B}\ \mathbf{I}, & \text{if } i = 0; \\ \mathbf{C}, & \text{if } i = 1; \\ \mathbf{B}\ \mathbf{C}\ (\mathbf{B}\ \mathbf{C}), & \text{if } i = 2; \\ \mathbf{B}\ \mathbf{C}\ (\mathbf{B}\ \mathbf{C}_{i-1}), & \text{if } i > 2; \end{cases} \\
a_i &= posi(i, \beta) - 1, \qquad \text{for } 1 \leq i \leq p; \\
\gamma_i &= \begin{cases} \beta_i, & \text{if } \beta_i \neq \mathbf{i}, \\ \mathbf{y}, & \text{if } \beta_i = \mathbf{i}, \end{cases} \qquad \text{for } 1 \leq i \leq m.
\end{aligned}$$

The definition of the $\mathbf{C}_i$ is based on that given by Smullyan [9, pp. 320 and 343], though he only defines $\mathbf{C}_i$, for $i \geq 2$. Note that $\mathbf{C}_i\ P\ Q\ R_1\ R_2\ \ldots\ R_i \to P\ R_1\ R_2\ \ldots\ R_i\ Q$, for $i \geq 1$. The order of $\mathbf{C}_i$ is $i + 2$, for $i \geq 0$. The following is an example of this translation function: $\mathit{trans}(\mathsf{innii}) = \mathbf{B}_2\ \mathbf{B}_2\ \mathbf{B}_2\ \mathbf{C}_0\ \mathbf{C}_3\ \mathbf{C}_4\ \mathsf{ynnyy\ I\ I\ I}$.

Proving that this translation is correct is quite convoluted. Before presenting the proof it is helpful to establish a number of lemmas.

The series of combinators H_i, for $i \geq 1$, are defined as follows:

$$H_i = \begin{cases} \mathbf{B}_2, & \text{if } i = 1; \\ \mathbf{B}_i\mathbf{B}_2H_{i-1}, & \text{if } i > 1. \end{cases}$$

It is straightforward to verify the following result

$$H_iP_1P_2\ldots P_i \to \mathbf{B}_2(\mathbf{B}_2(\ldots(\mathbf{B}_2(\mathbf{B}_2P_1P_2)P_3)\ldots)P_{i-1})P_i), \tag{3}$$

in which there are i occurrences of $\mathbf{B}_2$. The following lemma can now be proved:

Lemma 3 *For $j \geq 1$, we have*

$$H_jP_1P_2\ldots P_{2j+3} \to P_1(P_2(\ldots(P_j(P_{j+1}P_{j+2}P_{j+3})P_{j+4})\ldots)P_{2j+2})P_{2j+3}. \tag{4}$$

Proof The proof is by induction on j. In the base case $j = 1$. We have from the definition of H_1 and the behaviour of $\mathbf{B}_2$:

$$H_1P_1P_2P_3P_4P_5 = \mathbf{B}_2P_1P_2P_3P_4P_5 \to P_1(P_2P_3P_4)P_5.$$

Thus the base case has been established.

In the inductive step, we have to prove that (4) holds on the assumption that it holds when j is replaced by $j - 1$. From the definition of H_i we have:

$$\begin{aligned} H_jP_1P_2\ldots P_{2j+3} &= \mathbf{B}_j\mathbf{B}_2H_{j-1}P_1P_2\ldots P_{2j+3} \\ &\to \mathbf{B}_2(H_{j-1}P_1P_2\ldots P_j)P_{j+1}\ldots P_{2j+3} \\ &\to H_{j-1}P_1P_2\ldots P_j(P_{j+1}P_{j+2}P_{j+3})P_{j+4}\ldots P_{2j+3} \\ &\to P_1(P_2(\ldots(P_j(P_{j+1}P_{j+2}P_{j+3})P_{j+4})\ldots)P_{2j+2})P_{2j+3}, \end{aligned}$$

by the inductive hypothesis. The result follows by induction. QED.

The series of combinators G_i, for $i \geq 1$, are defined as follows:

$$G_i = \begin{cases} \mathbf{B}_2, & \text{if } i = 1; \\ G_{i-1}\underbrace{\mathbf{B}_2\mathbf{B}_2\ldots\mathbf{B}_2}_{i \text{ times}}, & \text{if } i > 1. \end{cases}$$

The following lemma can now be proved:

Lemma 4 *For $j \geq 1$, we have that*

$$G_jP_1P_2\ldots P_{2j+3} \to P_1(P_2(\ldots(P_j(P_{j+1}P_{j+2}P_{j+3})P_{j+4})\ldots)P_{2j+2})P_{2j+3}.$$

Proof The proof is by induction on j. In the base case, when $j = 1$, we have:

$$G_1P_1P_2P_3P_4P_5 = \mathbf{B}_2P_1P_2P_3P_4P_5 \to P_1(P_2P_3P_4)P_5.$$

Thus the base case has been established.

In the inductive step, we note that

$$\begin{aligned} G_jP_1P_2\ldots P_{2j+3} &= G_{j-1}\underbrace{\mathbf{B}_2\mathbf{B}_2\ldots\mathbf{B}_2}_{j \text{ times}}P_1P_2\ldots P_{2j+3} \\ &\to \mathbf{B}_2(\mathbf{B}_2(\ldots(\mathbf{B}_2(\mathbf{B}_2P_1P_2)P_3)\ldots)P_j)P_{j+1}\ldots P_{2j+3}, \end{aligned}$$

where there are j occurrences of $\mathbf{B}_2$, by the inductive hypothesis,

$$\to P_1(P_2(\ldots(P_j(P_{j+1}P_{j+2}P_{j+3})P_{j+4})\ldots)P_{2j+2})P_{2j+3},$$

by (3) and Lemma 3. The result follows by induction. QED.

Lemma 5 *Let* $0 \le a_1 < a_2 < \ldots < a_t$. *Then*

$$\begin{aligned} &\mathbf{C}_{a_1}(\mathbf{C}_{a_2}(\ldots(\mathbf{C}_{a_{t-1}}(\mathbf{C}_{a_t}MP_1)P_2)\ldots)P_{t-1})P_tP_{t+1}\ldots P_{a_t+1} \\ &\to MP_{f(1)}P_{f(2)}\ldots P_{f(t)}P_{f(t+1)}\ldots P_{f(a_t+1)}, \end{aligned}$$

where

$$f(i) = \begin{cases} i+t, & i < a_1+1; \\ t, & i = a_1+1; \\ i+(t-1), & a_1+1 < i < a_2+1; \\ t-1, & i = a_2+1; \\ i+(t-2), & a_2+1 < i < a_3+1; \\ t-2, & i = a_3+1; \\ \quad\vdots & \\ i+1, & a_{t-1}+1 < i < a_t+1; \\ 1, & i = a_t+1; \\ i, & a_t+1 < i. \end{cases}$$

Proof The proof is by induction on t. In the base case, when $t = 1$, we have:

$$\mathbf{C}_{a_1}MP_1P_2\ldots P_{a_1+1} \to MP_{f(1)}P_{f(2)}\ldots P_{f(a_1+1)},$$

where

$$f(i) = \begin{cases} i+1, & i < a_1+1; \\ 1, & i = a_1+1; \\ i, & a_1+1 < i. \end{cases}$$

This accords with the behaviour of $\mathbf{C}_{a_1}$ given above. Thus the base case has been established.

In the inductive step, we begin by noting that by the inductive hypothesis we have:

$$\begin{aligned}&\mathbf{C}_{a_1}(\mathbf{C}_{a_2}(\ldots(\mathbf{C}_{a_{t-1}}(\mathbf{C}_{a_{t-1}}NQ_1)Q_2)\ldots)Q_{t-1})Q_tQ_{t+1}\ldots Q_{a_{t-1}+1}\\ \rightarrow\ &NQ_{\phi(1)}Q_{\phi(2)}\ldots Q_{\phi(t)}Q_{\phi(t+1)}\ldots Q_{\phi(a_{t-1}+1)},\end{aligned}$$

where

$$\phi(i) = \begin{cases} i+(t-1), & i < a_1+1;\\ t-1, & i = a_1+1;\\ i+(t-2), & a_1+1 < i < a_2+1;\\ t-2, & i = a_2+1;\\ i+(t-3), & a_2+1 < i < a_3+1;\\ t-3, & i = a_3+1;\\ \quad\vdots & \\ i+1, & a_{t-2}+1 < i < a_{t-1}+1;\\ 1, & i = a_{t-1}+1;\\ i, & a_{t-1}+1 < i. \end{cases}$$

Let $N = \mathbf{C}_{a_t}MP_1$ and $P_i = Q_{i-1}$, for $i > 1$. Then

$$\begin{aligned}&NQ_{\phi(1)}Q_{\phi(2)}\ldots Q_{\phi(t)}Q_{\phi(t+1)}\ldots Q_{\phi(a_{t-1}+1)}\\ =\ &\mathbf{C}_{a_t}MP_1P_{\phi(1)+1}P_{\phi(2)+1}\ldots P_{\phi(a_{t-1}+1)+1}\\ =\ &\mathbf{C}_{a_t}MP_{\psi(1)}P_{\psi(2)}\ldots P_{\psi(t)}P_{\psi(t+1)}\ldots P_{\psi(a_{t-1}+2)},\end{aligned}$$

where $\psi(1) = 1$ and $\psi(i) = \phi(i-1)+1$, for $i > 1$,

$$\rightarrow MP_{\theta(1)}P_{\theta(2)}\ldots P_{\theta(a_t+2)}.$$

where

$$\theta(i) = \begin{cases} \psi(i+1), & i < a_t+1;\\ \psi(1), & i = a_t+1;\\ \psi(i), & a_t+1 < i. \end{cases}$$

Simplifying, this becomes:

$$\theta(i) = \begin{cases} \phi(i)+1, & i < a_t+1;\\ 1, & i = a_t+1;\\ \phi(i-1)+1, & a_t+1 < i. \end{cases}$$

Spelt out in full, this is:

$$\theta(i) = \begin{cases} i+t, & i < a_1 + 1; \\ t, & i = a_1 + 1; \\ i + (t-1), & a_1 + 1 < i < a_2 + 1; \\ t - 1, & i = a_2 + 1; \\ \quad \vdots & \\ i + 2, & a_{t-2} + 1 < i < a_{t-1} + 1 \\ 2, & i = a_t + 1; \\ i + 1, & a_{t-1} + 1 < i < a_t + 1; \\ 1, & i = a_t + 1; \\ i, & a_t + 1 < i. \end{cases}$$

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

Proposition 6 *The translation function trans is correct in the sense that if*

$$\beta\ P_1\ P_2\ \ldots\ P_n \to Q_1\ Q_2\ \ldots\ Q_m,$$

where the Q_i, for $1 \leq i \leq m$, are as given above, then

$$\mathit{trans}(\beta)\ P_1\ P_2\ \ldots\ P_n \to Q_1\ Q_2\ \ldots\ Q_m.$$

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

$$\mathit{trans}(\beta_1\beta_2\ldots\beta_m)\ P_1\ P_2\ \ldots\ P_n \to Q_1\ Q_2\ \ldots\ Q_m,$$

where $n = m + 1 - ic(\beta_1\beta_2\ldots\beta_m)$ and, for $1 \leq i \leq m$, we have that

$$Q_i = \begin{cases} P_n, & \text{if } \beta_i = \mathbf{i}; \\ P_{f(i)}\ P_n, & \text{if } \beta_i = \mathbf{y}; \\ P_{f(i)}, & \text{if } \beta_i = \mathbf{n}; \end{cases}$$

where $f(i) = i - ic(\beta_1\beta_2\ldots\beta_i)$. From the definition of *trans* we have that

$$\begin{aligned} & \mathit{trans}(\beta_1\beta_2\ldots\beta_m)\ P_1\ P_2\ \ldots\ P_n \\ = & \underbrace{\mathbf{B}_2\ \mathbf{B}_2\ \ldots\ \mathbf{B}_2}_{q \text{ times}}\ \mathbf{C}_{a_1}\ \mathbf{C}_{a_2}\ \ldots\ \mathbf{C}_{a_p}\ \gamma_1\gamma_2\ldots\gamma_m\ \underbrace{\mathbf{I}\ \mathbf{I}\ \ldots\ \mathbf{I}}_{p \text{ times}}\ P_1\ P_2\ \ldots\ P_n \\ \to & \mathbf{C}_{a_1}(\mathbf{C}_{a_2}(\ldots(\mathbf{C}_{a_{p-1}}(\mathbf{C}_{a_p}\gamma_1\gamma_2\ldots\gamma_m\ \mathbf{I})\mathbf{I})\ldots)\mathbf{I})\mathbf{I}P_1\ P_2\ \ldots\ P_n, \end{aligned}$$

by Lemma 4,

$$\to \gamma_1\gamma_2\ldots\gamma_m\ R_{f(1)}\ R_{f(1)}\ \ldots\ R_{f(m+1)},$$

where

$$f(i) = \begin{cases} i+p, & i < a_1 + 1; \\ p, & i = a_1 + 1; \\ i+(p-1), & a_1 + 1 < i < a_2 + 1; \\ p-1, & i = a_2 + 1; \\ i+(p-2), & a_2 + 1 < i < a_3 + 1; \\ p-2, & i = a_3 + 1; \\ \quad \vdots & \\ i+1, & a_{p-1} + 1 < i < a_p + 1; \\ 1, & i = a_p + 1; \\ i, & a_p + 1 < i. \end{cases}$$

Here $R_i = \mathbf{l}$, for $1 \leq i \leq p$, and $R_{i+p} = P_i$, for $1 \leq i \leq n$. We now have that

$$\gamma_1\gamma_2 \ldots \gamma_m \; R_{f(1)} \; R_{f(2)} \; \ldots \; R_{f(m+1)} \rightarrow S_1 \; S_2 \; \ldots S_m,$$

where

$$S_i = \begin{cases} R_{f(i)}, & \text{if } \gamma_i = \mathbf{n}; \\ R_{f(i)} \; R_{f(m+1)}, & \text{if } \gamma_i = \mathbf{y}. \end{cases}$$

In order to prove the correctness of the translation we need to show that this is equivalent to the following:

$$\beta_1\beta_2 \ldots \beta_m \; P_1 \; P_2 \; \ldots \; P_n \rightarrow Q_1 \; Q_2 \; \ldots \; Q_m,$$

where $n = m + 1 - ic(\beta_1\beta_2 \ldots \beta_m)$ and, for $1 \leq i \leq m$, we have that

$$Q_i = \begin{cases} P_n, & \text{if } \beta_i = \mathbf{i}; \\ P_{g(i)} \; P_n, & \text{if } \beta_i = \mathbf{y}; \\ P_{g(i)}, & \text{if } \beta_i = \mathbf{n}; \end{cases}$$

where $g(i) = i - ic(\beta_1\beta_2 \ldots \beta_i)$.

When $\beta_i = \mathbf{i}$, then $Q_i = P_n$. Because $\beta_i = \mathbf{i}$, we have that $a_i = posi(i, \beta) - 1$. When that happens, looking at the definition of f, we have that $f(i) = p - i$. Therefore, $R_{f(i)} = R_{p-i} = \mathbf{l}$. Also, when $\beta_i = \mathbf{i}$, then $\gamma_i = \mathbf{y}$ and $S_i = R_{f(i)} \; R_{f(m+1)} = \mathbf{l} \; P_n$, as $m + 1 = p + n$. Thus, $S_i \rightarrow Q_i$.

When $\beta_i = \mathbf{n}$, we have that $Q_i = P_{g(i)}$. Because $\beta_i \neq \mathbf{i}$, $i \neq a_j + 1$, for any j such that $1 \leq j \leq p$. Therefore, either $i < a_1 + 1$ or $a_j + 1 < i < a_{j+1} + 1$, for some j such that $1 \leq j \leq p - 1$, or $a_p + 1 < i$. I will look at each of these three cases in turn.

When $i < a_1 + 1 = posi(1, \beta)$, then $ic(\beta_1) = 0$ and $g(i) = i$. Thus, $Q_i = P_i = R_{i+p}$. We also have that $S_i = R_{f(i)} = R_{i+p}$, because $i < a_1 + 1$. Therefore, $Q_i = S_i$.

When $posi(j, \beta) = a_j + 1 < i < a_{j+1} + 1 = posi(j + 1, \beta)$, then we have that $ic(\beta_i) = j$ and $g(i) = i - j$. This means that $Q_i = P_{i-j} = R_{i-j+p}$. We also have that $S_i = R_{f(i)} = R_{i+p-j}$. Therefore, $Q_i = S_i$.

In this algorithm E is any term such that $x \notin FV(E)$ and P_1 is an atom. The clauses of this algorithm have to be applied in the order in which they occur here.

$$[x]\ E\ x = E,$$
$$[x]\ P_1\ P_2\ \ldots\ P_m = \beta_1\beta_2\ldots\beta_m\ Q_1\ Q_2\ \ldots\ Q_n,$$

where $n = m - ac(x, P_1\ P_2\ \ldots\ P_m)$ and, for $1 \leq i \leq m$,

$$\begin{array}{lll} \beta_i = \mathsf{i}, & & \text{if } P_i = x; \\ \beta_i = \mathsf{y} \quad \text{and} \quad Q_{f(i)} = [x]\ P_i, & & \text{if } P_i \neq x, \text{ but } x \in FV(P_i); \\ \beta_i = \mathsf{n} \quad \text{and} \quad Q_{f(i)} = P_i, & & \text{if } x \notin FV(P_i); \end{array}$$

where $f(i) = i - ac(x, P_1\ P_2\ \ldots\ P_i)$.

Figure 2: Algorithm (**yin**).

When $posi(p, \beta) = a_p + 1 < i$, then $ic(\beta_i) = p$ and $g(i) = i - p$. This means that $Q_i = P_{i-p} = R_i$. We also have that $S_i = R_{f(i)} = R_i$. Therefore, $Q_i = S_i$.

When $\beta_i = \mathsf{y}$, the reasoning is similar to that employed when $\beta_i = \mathsf{n}$. In addition, however, we need to note that $R_{f(m+1)} = R_{m+1} = R_{n+p} = P_n$.

Thus, the translation function *trans* has been shown to be correct. QED.

4.2 Algorithm (yin)

Algorithm (**yin**) is shown in Figure 2. An example of its application should make its operation clear:

$$\begin{aligned} & [x]\ x\ (y\ z)\ (z\ y\ x)\ (z\ (x\ y)) \\ = {} & \mathsf{inyy}\ (y\ z)\ ([x]\ z\ y\ x)\ ([x]\ z\ (x\ y)) \\ = {} & \mathsf{inyy}\ (y\ z)\ (z\ y)\ (\mathsf{ny}\ z\ ([x]\ x\ y)) \\ = {} & \mathsf{inyy}\ (y\ z)\ (z\ y)\ (\mathsf{ny}\ z\ (\mathsf{in}\ y)). \end{aligned}$$

Note that algorithm (**yin**) is not well-behaved under self-composition as that has been explained so far in this paper. However, if the input to algorithm (**yin**) has the form $KQ_1Q_2\ldots Q_i$, then the output has the form $KQ_1Q_2\ldots Q_j$, where $0 \leq j \leq i$.

Proposition 7 *Let P be any term. Then algorithm (**yin**) has the following properties:*

*(a) Algorithm (**yin**) terminates.*

*(b) Algorithm (**yin**) has the property that $([x]P)x \to P$.*

*(c) Let algorithm (**yin**$\backslash c$) be the same as (**yin**) except that the clause $[x]\ E\ x = E$ has been deleted. Then using algorithm (**yin**$\backslash c$) we have that*

$$\#([x]P) = 1 + \#P + rpx(x, P) - tac(x, P).$$

(d) Let $[x]_{\mathsf{yin}}\ P$ *represent the result of applying algorithm (*yin*) to* P *and let* $[x]_{\mathsf{yn}}\ P$ *represent the result of applying (*yn*) to* P*. Then* $\#([x]_{\mathsf{yin}}\ P) \leq \#([x]_{\mathsf{yn}}\ P)$.

(e) Using algorithm ($\mathsf{yin}\backslash c$*) we have that*

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

Proof The proofs of parts (a), (b), (c) and (d) are all by induction on the value of $rpx(x, P)$. Part (e) can be proved by induction on a using part (c) and the fact that $rpx(y, [x]P) = rpx(y, P)$, when $(\mathsf{yin}\backslash c)$ is being used.

The proofs are all fairly straightforward and so only that for part (c) will be given here.

Proof of part (c): The proof is by induction on the value of $rpx(x, P)$. In the base case $rpx(x, P) = 0$. Let $P = P_1\ P_2\ \ldots\ P_m$. We thus have:

$$[x]\ P_1\ P_2\ \ldots\ P_m = \beta_1\beta_2\ldots\beta_m\ Q_1\ Q_2\ \ldots\ Q_n,$$

where $n = m - ac(x, P)$ and the Q_i, for $1 \leq i \leq m$, are as shown in Fig. 2. Therefore,

$$\begin{aligned}\#([x]\ P) &= 1 + m - ac(x, P)\\ &= 1 + \#P - tac(x, P) + rpx(x, P),\end{aligned}$$

because, in this case, $ac(x, P) = tac(x, P)$ and $rpx(x, P) = 0$. Thus, the base case has been established.

In order to establish the inductive step, first let $P = P_1\ P_2\ \ldots\ P_m$. We thus have:

$$[x]\ P_1\ P_2\ \ldots\ P_m = \beta_1\beta_2\ldots\beta_m\ Q_1\ Q_2\ \ldots\ Q_n,$$

where $n = m - ac(x, P)$ and the Q_i, for $1 \leq i \leq m$, are as shown in Fig. 2. Therefore,

$$\begin{aligned}\#([x]\ P) &= 1 + \sum_{i=1}^{m}\textbf{if } P_i = x \textbf{ then } 0 \textbf{ else}\\ &\qquad\qquad \textbf{if } P_i \neq x \textbf{ and } x \in FV(P_i) \textbf{ then } \#([x]P_i) \textbf{ else}\\ &\qquad\qquad \textbf{if } x \notin FV(P_i) \textbf{ then } \#P_i\\ &= 1 + \sum_{i=1}^{m}\textbf{if } P_i = x \textbf{ then } 0 \textbf{ else}\\ &\qquad\qquad \textbf{if } P_i \neq x \textbf{ and } x \in FV(P_i) \textbf{ then}\\ &\qquad\qquad\quad 1 + \#P_i + rpx(x, P_i) - tac(x, P_i) \textbf{ else}\\ &\qquad\qquad \textbf{if } x \notin FV(P_i) \textbf{ then } \#P_i,\end{aligned}$$

by the inductive hypothesis,

$$\begin{aligned}&= 1 + \#P - ac(x, P) +\\ &\qquad\qquad \sum_{i=1}^{m}\textbf{if } P_i \neq x \textbf{ and } x \in FV(P_i) \textbf{ then}\\ &\qquad\qquad\quad 1 + rpx(x, P_i) - tac(x, P_i) \textbf{ else } 0,\end{aligned}$$

because $\#P = \sum_{i=1}^{m} \#P_i$, but we need to subtract 1 each time $P_i = x$ in order to obtain the number we need,

$$= 1 + \#P - tac(x, P) + rpx(x, P),$$

because $tac(x, P) = ac(x, P) + \sum_{i=1}^{m}$ **if** $P_i \neq x$ **then** $tac(x, P_i)$ **else** 0. The result follows by induction. QED.

5 Conclusion

Although a great deal of effort has been put into devising bracket abstraction algorithms since Turner published his in 1979, that algorithm still remains one of the best for the purpose of implementing a functional language. In addition, Turner's work on bracket abstraction has proved to be very fruitful. Bunder [2], for example, analyses the role of optimisations in Turner's algorithm and improves that algorithm by adding a large number of additional optimisations. In this paper I have focused on another of Turner's ideas, namely that of an algorithm's being well-behaved under self-composition. By employing Stevens's idea of representing combinators iconically [10] it is possible to devise algorithms that are always well-behaved under self-composition, no matter how many primal components the terms to which they are applied have. Algorithm (**yn**) is such an algorithm. That algorithm, however, has its own limitations. The abstracts produced can contain a large number of occurrences of the combinator **I**. These can be eliminated by introducing **yin** representations. Algorithm (**yin**) makes use of these. Although the resulting algorithm is not, strictly speaking, well-behaved under self-composition, it always produces abstracts of the form $KQ_1Q_2 \ldots Q_i$, for some i. Furthermore, both algorithms presented in this paper never produce abstracts longer than those produced by algorithm (C) and both of them have the property that $([x]X)x \to X$.

In recent years research effort in the implementation of functional programming languages has tended to be directed towards super-combinator methods (introduced by Hughes [5] in the early 1980s) and research on combinator-based techniques has been partially eclipsed by the success of those super-combinator methods. I believe that the full potential of combinator-based techniques has yet to be realised. Bunder [2], Stevens [10] and the work presented in this paper show that there are still interesting ways of improving bracket abstraction algorithms. I hope that this work will encourage others to look for new ideas that can be used to further improve bracket abstraction algorithms and deepen our understanding of those algorithms.

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

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

[3] Haskell B. Curry and Robert Feys. *Combinatory Logic*, volume 1. North-Holland, Amsterdam, 1958.

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

[5] John Hughes. Graph reduction with super-combinators. Technical Monograph PRG–28, Programming Research Group, Oxford University Computing Laboratory, 1982.

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

[7] J. R. Kennaway. The complexity of a translation of λ-calculus to combinators. Internal Report CSA/13/1984, School of Information Systems, University of East Anglia, 1984.

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

[9] Raymond M. Smullyan. *Diagonalization and Self-reference*, volume 27 of *Oxford Logic Guides*. Oxford University Press, Oxford, 1994.

[10] 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.

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