1990

Some improvements to Turner's algorithm for bracket abstraction

Martin W. Bunder
University of Wollongong, mbunder@uow.edu.au

Publication Details

Bunder, M. W. (1990). Some improvements to Turner's algorithm for bracket abstraction. The Journal of Symbolic Logic, 55 (2), 656-669.

Some improvements to Turner's algorithm for bracket abstraction

Abstract
A computer handles A-terms more easily if these are translated into combinatory terms. This translation process is called bracket abstraction. The simplest abstraction algorithm-the (fab) algorithm of Curry (see Curry and Feys [6])-is lengthy to implement and produces combinatory terms that increase rapidly in length as the number of variables to be abstracted increases.

Keywords
turner, improvements, algorithm, abstraction, bracket

Publication Details
Bunder, M. W. (1990). Some improvements to Turner's algorithm for bracket abstraction. The Journal of Symbolic Logic, 55 (2), 656-669.

THE JOURNAL OF SYMBOLIC LOGIC
Volume 55, Number 2, June 1990

SOME IMPROVEMENTS TO TURNER'S ALGORITHM FOR BRACKET ABSTRACTION

M. W. BUNDER

Introduction. A computer handles λ-terms more easily if these are translated into combinatory terms. This translation process is called bracket abstraction. The simplest abstraction algorithm—the (fab) algorithm of Curry (see Curry and Feys [6])—is lengthy to implement and produces combinatory terms that increase rapidly in length as the number of variables to be abstracted increases.

There are several ways in which these problems can be alleviated:

(1) A change in order of the clauses in the algorithm so that (f) is performed as a last resort.

(2) The use of an extra clause (c), appropriate to $\beta\eta$ reduction.

(3) The introduction of a finite number of extra combinators.

The original 1924 form of bracket abstraction of Schönfinkel [17], which in fact predates λ-calculus, uses all three of these techniques; all are also mentioned in Curry and Feys [6].

A technique employed by many computing scientists (Turner [20], Peyton Jones [16], Oberhauser [15]) is to use the (fab) algorithm followed by certain "optimizations" or simplifications involving extra combinators and sometimes special cases of (c).

Another is either to allow a fixed infinite set of (super-) combinators (Abdali [1], Kennaway and Sleep [10], Krishnamurthy [12], Tonino [19]) or to allow new combinators to be defined one by one during the abstraction process (Hughes [7] and [8]).

A final method encodes the variables to be abstracted as an n-tuple—this requires only a finite number of combinators (Curien [5], Statman [18]).[1]

A measure of the efficiency of an abstraction algorithm was first introduced by Kennaway in [9] as an upper bound of the length of the obtained combinatory term, as a function of the length of the original term and the number of variables to be abstracted. Mulder in [14] used these methods and experiments with many special cases, to compare the efficiency of the main algorithms listed above. The

Received December 19, 1988; revised May 8, 1989.

[1] Curien's algorithm can be made more efficient if it is generalised to allow an infinite set of combinators (see Lins [13]).

0022-4812/90/5502-0018/$02.40

algorithm of Statman [18] came out as the most efficient in the limiting case, but showed up as almost the worst in a number of reasonably simple special cases. Turner's algorithm [20] was generally the best in these cases and was in fact Mulder's choice overall.

Hughes in [8] used a program to generate terms to be abstracted, and found that for relatively short terms Turner's algorithm was far superior to his.

Burton in [4] also mentions advantages of Turner's algorithm over the supercombinators of Abdali [1].

It is therefore clear that Turner's algorithm remains of substantial interest.

In this paper, firstly we note that what Turner describes as "the improved algorithm of Curry", on which his own is based, is in fact not equivalent to any of Curry's algorithms. Turner's abstracts lack a basic property possessed by all of Curry's as well as many others.

Secondly we give methods whereby Turner's algorithm (as well as others) can be more efficiently implemented, while providing simpler abstracts.

Some definitions. The variables $x_1, x_2, \ldots$ and $x, y, z, \ldots$ are *primitive λ-terms.*

If X and Y are λ-terms and x is a variable, then (XY) and (λxX) are *λ-terms.*

(XY) is formed by *application*, λxX by *abstraction.*

The essential postulate of the λ-calculus is:

$$(\beta) \qquad (\lambda x.X)Y = [Y/x]X$$

(for details see Curry and Feys [6]).

The above variables are also *combinatory terms*, as are certain constants among them ("combinators").

If X and Y are combinatory terms, so is (XY).

Terms constructed as the λ-terms but with constants including combinators as primitive λ-terms we will refer to as *λ-combinatory terms.*

In both λ-calculus and combinatory logic we abbreviate terms, formed by application, by associating to the left. $\lambda x.XY$ abbreviates $\lambda x(XY)$, and $\lambda x_1 \cdots x_n.Y$ abbreviates $\lambda x_1(\lambda x_2(\cdots(\lambda x_n Y)\cdots))$.

The Curry and Schönfinkel algorithms. Given that we have among the combinatory constants combinators **S**, **K** and **I** with the properties:[2]

$$(\mathbf{S}) \qquad \mathbf{S}XYZ = XZ(YZ),$$

$$(\mathbf{K}) \qquad \mathbf{K}XY = X,$$

$$(\mathbf{I}) \qquad \mathbf{I}X = X.$$

$[x]X$, the combinatory term corresponding to λxX, can be defined recursively, by Curry's (fab) algorithm, as follows:

$$(\mathrm{f}) \qquad [x].XY \equiv \mathbf{S}([x]X)([x]Y);$$

$$(\mathrm{a}) \qquad [x]Z \equiv \mathbf{K}Z,$$

[2] **S** and **K** are normally primitive constants. **I** may be taken as primitive constant or can be defined as **SKK**. All other combinators can be formed by application from **S** and **K**.

where x is not in Z; and

(b) $$[x]x \equiv \mathbf{I}.$$

Note that when one of several λ-abstracts in a λ-term has been replaced by its bracket form, we obtain a λ-combinatory term. Further translations therefore have to be made to such terms.

As we will be dealing with various algorithms, we will write $\lambda^* x$ for $[x]$, where the * will denote the algorithm being used. We will use conventions abbreviating terms involving [ ] or λ^* similar to those for λ.

Note that in the algorithm above (and any algorithm) the clauses must be used in the order given.

The (fab) algorithm is not efficient and produces extremely long abstracts, for example:

$$\lambda^{\mathrm{fab}} x_1 x_2.\mathbf{K}x_1 \equiv \mathbf{S}[\mathbf{S}(\mathbf{KS})(\mathbf{S}(\mathbf{KK})(\mathbf{KK}))][\mathbf{S}(\mathbf{KK})\mathbf{I}].$$

If the order of the clauses is changed to (abf), we have

$$\lambda^{\mathrm{abf}} x_1 x_2.\mathbf{K}x_1 \equiv \mathbf{S}(\mathbf{KK})(\mathbf{S}(\mathbf{KK})\mathbf{I}).$$

The equality generated by (**S**), (**K**), (**I**), further axioms for other primitive combinators (if any) (such as (**B**) and (**C**) below), and the usual equivalence and substitution properties of equality, is called *weak equality* and is denoted by $=_{\mathrm{w}}$.

If equality satisfies the additional rule[3]

(ζ) $$\text{If } Xx = Yx \text{ and } x \notin \mathrm{FV}(XY), \text{ then } X = Y,$$

it is called *strong* or *$\beta\eta$ equality*. The two abstracts of $\mathbf{K}x_1$ derived above are clearly equal in this sense.

Curry produced more efficient algorithms still, by including (some of) the clauses

(c) $$[x].Xx \equiv X, \qquad \text{if } x \notin \mathrm{FV}(X),$$

(d) $$[x].XY \equiv \mathbf{B}X([x]Y), \qquad \text{if } x \notin \mathrm{FV}(X),$$

(e) $$[x].XY \equiv \mathbf{C}([x]X)Y, \qquad \text{if } x \notin \mathrm{FV}(Y),$$

where[4]

(**B**) $$\mathbf{B}UVW = U(VW),$$

(**C**) $$\mathbf{C}UVW = UWV.$$

We then have

$$\lambda^{\mathrm{abcf}} x_1 x_2.\mathbf{K}x_1 \equiv \mathbf{S}(\mathbf{KK})\mathbf{K}$$

and

$$\lambda^{\mathrm{abcdef}} x_1 x_2.\mathbf{K}x_1 \equiv \mathbf{BKK}.$$

[3] $\mathrm{FV}(X)$ is the set of variables in the combinatory term X.

[4] **B** and **C** can be extra primitives constants or can be defined as **S(KS)K** and **S(BBS)(KK)** respectively.

If **B** and **C** are defined in terms of **K** and **S**, these algorithms are equivalent and the abstraction is usually written as $\lambda\eta$. The algorithm (abcdef) is effectively that of Schönfinkel [17].

Turner's "improved algorithm of Curry". Turner represents his algorithm as an extension of what he calls "the improved algorithm of Curry", by which he clearly means (abcdef). These are in fact not equivalent.

Turner achieves the improvement by adding to the (fab) algorithm the following "optimizations" to be used, if possible, after any abstraction:

(1) $$\mathbf{S}(\mathbf{K}X)(\mathbf{K}Y) \to \mathbf{K}(XY),$$

(2) $$\mathbf{S}(\mathbf{K}X)\mathbf{I} \to X,$$

(3) $$\mathbf{S}(\mathbf{K}X)Y \to \mathbf{B}XY,$$

(4) $$\mathbf{S}X(\mathbf{K}Y) \to \mathbf{C}XY.$$

(If more than one optimization is applicable, the earlier one takes precedence.)

Using only (1), we can prove:

(5) $$\text{if } x \notin \mathrm{FV}(X) \text{ then } [x]X \equiv \mathbf{K}X,$$

so the addition of (1) strengthens λ^{fab} to λ^{abf}.

The addition of (1) and (2) allows the proof of[5]

(6) $$\text{if } x \notin \mathrm{FV}(X) \text{ then } [x].Xx \equiv X,$$

so it may seem that this gives λ^{abcf} (or λ^{η}) and the addition of (3) and (4) gives $\lambda^{\mathrm{abcdef}}$. However, Turner's version of λ^{abcf} (and $\lambda^{\mathrm{abcdef}}$) fails to satisfy a property satisfied by all Curry algorithms, namely[6]

(7) $$([x]X)x > X.$$

Here is an example:

$$(\lambda^{\mathrm{abcf}}x.\mathbf{KS}x(\mathbf{KS}x))x \equiv \mathbf{S}(\mathbf{KS})(\mathbf{KS})x > \mathbf{KS}x(\mathbf{KS}x),$$

while, naming Turner's version of λ^{abcf}, $\lambda^{\mathrm{abcf(T)}}$, we have using (1):

$$(\lambda^{\mathrm{abcf(T)}}x.\mathbf{KS}x(\mathbf{KS}x))x \equiv \mathbf{K}(\mathbf{SS})x \not> \mathbf{KS}x(\mathbf{KS}x).$$

Of course, the two abstracts are what Turner calls "extensionally equal", in that

$$\mathbf{S}(\mathbf{KS})(\mathbf{KS})x =_{\mathrm{w}} \mathbf{K}(\mathbf{SS})x.$$

However,

$$\lambda^{\mathrm{abcdef}}x_1.x_1(\mathbf{S}(\mathbf{K}x_2)(\mathbf{K}x_3)). \equiv \mathbf{CI}(\mathbf{S}(\mathbf{K}x_2)(\mathbf{K}x_3))$$

[5] An equality relation that satisfies (5) and or (6) is of course a stronger relation than the weak equality ($=_{\mathrm{w}}$) that satisfies only (**I**), (**S**), (**K**), and the usual substitution properties.

[6] The postulates of $>$ are (**I**), (**K**) and (**S**) (and, if needed, (**B**) and (**C**)) with $>$ for $=$, transitivity, and "if $U > V$ then $ZU > ZV$ and $UZ > VZ$".

is not equal to

$$\lambda^{\mathrm{abcdef(T)}}x_1.x_1(\mathbf{S}(\mathbf{K}x_2)(\mathbf{K}x_3)) \equiv \mathbf{CI}(\mathbf{K}(x_2x_3))$$

(which is in fact the Turner abstract) even in this sense. (They are of course $\beta\eta$-equal.)

Turner's algorithm. This uses "new combinators" $\mathbf{S}'$, $\mathbf{B}'$ and $\mathbf{C}'$, which are such that[7]

$$\mathbf{S}'XYZT = X(YT)(ZT),$$
$$\mathbf{B}'XYZT = XY(ZT),$$
$$\mathbf{C}'XYZT = X(YT)Z.$$

The Turner algorithm is then his "improved algorithm of Curry" with the following additional optimizations:

(8) $$\mathbf{S}(\mathbf{B}XY)Z \to \mathbf{S}'XYZ,$$

(9) $$\mathbf{B}(XY)Z \to \mathbf{B}'XYZ,$$

(10) $$\mathbf{C}(\mathbf{B}XY)Z \to \mathbf{C}'XYZ.$$

We will write the Turner abstraction operator as $\lambda^{\mathrm{T}}x$.

Note that a counterpart to this algorithm that satisfies (7) has Curry's clauses (abcdef) with between (c) and (d):

(d′) $[x].XYZ \equiv \mathbf{B}'XY([x]Z)$, if $x \notin \mathrm{FV}(XY)$,

(e′) $[x].XYZ \equiv \mathbf{C}'X([x]Y)Z$, if $x \notin \mathrm{FV}(XZ)$,

(f′) $[x].XYZ \equiv \mathbf{S}'X([x]Y)([x]Z)$, if $x \notin \mathrm{FV}(X)$.

This (abcd′e′f′def) algorithm is rather more efficient than Turner's. Turner's algorithm gives

$$[x_4].x_1(x_2(x_3x_4)) = \mathbf{B}x_1(\mathbf{B}x_2(\mathbf{B}x_3\mathbf{I}))$$

after 7 abstraction operations and 3 optimizations. The extended Curry algorithm gives

$$[x_4].x_1(x_2(x_3x_4)) = \mathbf{B}x_1(\mathbf{B}x_2x_3)$$

after only 3 abstraction operations. This is one of the two simplest possible forms of this abstract.

Our new algorithm, introduced below, is based on (abcd′e′f′def) and some optimizations.

An extension of λ^{T}. Turner's algorithm, rather than finding as $[x]X$ a term Y such that $Yx > X$, simply finds a term so that $Yx =_{\beta\eta} X$. We will find simpler such Y's by faster means.

[7] $\mathbf{S}'$ is exactly the Φ of Curry and Feys [6], defined there as $\mathbf{B}(\mathbf{BS})\mathbf{B}$. $\mathbf{B}'$ can be defined by $\mathbf{BB}$ and $\mathbf{C}'$ by $\mathbf{B}(\mathbf{BC})\mathbf{B}$.

If we assume as new optimizations

(11) $\mathbf{I}X \to X,$

(12) $\mathbf{K}XY \to X,$

(13) $\mathbf{B}XYZ \to X(YZ),$

(14) $\mathbf{C}XYZ \to XZY,$

(15) $\mathbf{S}XYx \to Xx(Yx),$

as well as the rules:

(μ) if $X \to Y$, then $ZX \to ZY,$

(ν) if $X \to Y$, then $XZ \to YZ,$

(τ) if $X \to Y$ and $Y \to Z$, then $X \to Z,$

each of Turner's optimizations can be derived using a single application of the rule

(ξ) if $X \to Y$ then $\lambda^{\mathrm{T}}x.X \to \lambda^{\mathrm{T}}x.Y.$

(ξ) simply generates some functionally (or β) equal terms.

(ξ) however does generate optimizations other than Turner's; for example,

$$\mathbf{S}(\mathbf{S}(\mathbf{KK})\mathbf{I})\mathbf{I} \to \mathbf{I}.$$

In [2] and [3], we found combinatory equations which allowed the proof of the admissibility of (ξ); with $=$ for $\to$ for various abstraction operators, this generalises the work in §6C of Curry and Feys [6]. We could attempt in a similar fashion to derive optimizations equivalent to the above form of (ξ), from which we could then perhaps generate possible shorter forms of any abstract. This approach, however, seems not to work, and many of the optimizations found using this approach did not seem useful for practical purposes.

What is often useful in evaluating $\lambda^{\mathrm{T}}x.X$ is to optimize first, using (11) to (14), and then to abstract. It is then often not necessary to optimize again.

For example, $\lambda^{\mathrm{T}}x.\mathbf{K}yx$ becomes by (fab):

$$\begin{aligned} \mathbf{S}(\mathbf{S}(\mathbf{KK}_{\prime}(\mathbf{K}y))\mathbf{I} &\to \mathbf{S}(\mathbf{K}(\mathbf{K}y))\mathbf{I} && \text{by (1)} \\ &\to \mathbf{K}y && \text{by (2).} \end{aligned}$$

Using (12) first, we have

$$\lambda^{*}x.\mathbf{K}yx \to \lambda^{*}x.y \equiv \mathbf{K}y.$$

Sometimes this optimization-first algorithm, which we call * for now, gives, rather than the Turner abstract, a simpler one (by a shorter method). For example:

$$\lambda^{\mathrm{T}}x.\mathbf{K}(\mathbf{K}x)x \equiv \mathbf{S}'\mathbf{KKI},$$

while

$$\lambda^{*}x.\mathbf{K}(\mathbf{K}x)x \to \lambda^{*}x.\mathbf{K}x \equiv \mathbf{S}(\mathbf{KK})\mathbf{I} \to \mathbf{K}.$$

The final optimization could have been saved if * included (c).

Using optimizations other than (11)–(14) before abstracting also can simplify the procedure and the abstract:

$$\lambda^{\mathrm{T}}x.\mathbf{S}(\mathbf{KI})(\mathbf{K}x) \equiv \mathbf{B}'\mathbf{S}(\mathbf{KI})\mathbf{K},$$

while

$$\begin{aligned}\lambda^*x.\mathbf{S}(\mathbf{KI})(\mathbf{K}x) &\to \lambda^*x.\mathbf{K}(\mathbf{I}x) \qquad \text{by (11)}\\ &\to \lambda^*x.\mathbf{K}x \equiv \mathbf{K}\end{aligned}$$

(assuming * includes (c)).

The new algorithm, outlined in the next section, involves a number of optimizations of this kind, followed by an abstraction step. Any λ-combinatory term can in this way be transformed into a combinatory term if our first abstraction steps apply to terms with no inner λ's. For example:

$$\lambda^*z.(\lambda^*x.y)z \equiv \lambda^*z.\mathbf{K}yz \equiv \mathbf{K}y.$$

We can, however, save ourselves an abstraction step by applying the β reduction. That is,

$$\lambda^*z.(\lambda^*x.y)z \to \lambda^*z.y \equiv \mathbf{K}y.$$

Sometimes, on the other hand, performing a β-reduction first will complicate the combinatory term obtained. For example:

$$(\lambda x^*.xxx)(\lambda^*x.xx) \to (\lambda x^*.xx)(\lambda x^*.xx)(\lambda x^*.xx)$$

and

$$(\lambda^{\mathrm{T}}x.xx)(\lambda^{\mathrm{T}}x.xx)(\lambda^{\mathrm{T}}x.xx) \equiv \mathbf{SII}(\mathbf{SII})(\mathbf{SII}),$$

while

$$(\lambda^{\mathrm{T}}x.xxx)(\lambda^{\mathrm{T}}x.xx) \equiv \mathbf{S}(\mathbf{SII})\mathbf{I}(\mathbf{SII}).$$

We therefore will allow only reductions that simplify a term.

For full flexibility we will also allow abstractions of terms with internal λ-abstracts that cannot be removed by β steps that simplify the term concerned.

The extended Turner algorithm. Our new algorithm, which we will call E, will use the (abcd′e′f′def) algorithm preceded by (11)–(14), Turner's optimizations (1), (2) and (4),

(16) $$\mathbf{S}(\mathbf{K}X) \to \mathbf{B}X,$$

(17) $$\mathbf{S}(\mathbf{B}XY) \to \mathbf{S}'XY,$$

(18) $$\mathbf{B}(XY) \to \mathbf{B}'XY,$$

(19) $$\mathbf{C}(\mathbf{B}XY) \to \mathbf{C}'XY,$$

the $\beta\eta$-optimizations:

(20) $$(\lambda^E x.x)T \to T,$$

(21) if $x \notin \mathrm{FV}(X)$ $$(\lambda^E x.X)T \to X,$$

(22) if $x \notin \mathrm{FV}(X)$, $(\lambda^E x.Xx) \to X$,

(23) if $x \notin \mathrm{FV}(X)$, $(\lambda^E x.XY)T \to X([T/x]Y)$

and x appears only once in Y;

(24) if $x \notin \mathrm{FV}(Y)$, $(\lambda^E x.XY)T \to ([T/x]X)Y$

and x appears only once in X; as well as several other optimizations:

(25) $\mathbf{B}X(\mathbf{K}Y) \to \mathbf{K}(XY)$,

(26) $\mathbf{B}X\mathbf{I} \to X$,

(27) $\mathbf{C}(\mathbf{K}X)Y \to \mathbf{K}(XY)$,

(28) $\mathbf{BI} \to \mathbf{I}$,

(29) $\mathbf{SK}X \to \mathbf{I}$,

(30) $\mathbf{S}(\mathbf{BK}X) \to \mathbf{K}X$,

(31) $\mathbf{S}(\mathbf{BK}X)Y \to X$,

(32) $\mathbf{S'K} \to \mathbf{K}$,

(33) $\mathbf{S'I} \to \mathbf{S}$,

(34) $\mathbf{B'I} \to \mathbf{B}$,

(35) $\mathbf{C'I} \to \mathbf{C}$,

(36) $\mathbf{C'K} \to \mathbf{K}$,

(37) $\mathbf{B'}XYZT \to XY(ZT)$,

(38) $\mathbf{C'}XYZT \to X(YT)Z$,

(39) $\mathbf{B'K}XYZ \to X$,

(40) $\mathbf{S'}(\mathbf{K}X)Y \to \mathbf{B}X$,

(41) $\mathbf{S'}X(\mathbf{K}Y) \to \mathbf{B'}XY$,

(42) $\mathbf{S'}XY(\mathbf{K}Z) \to \mathbf{C'}XYZ$.

We assume the order of these to be as given.

We will call this complete list of optimizations the *E-optimizations*.

Note that the **S** reduction rule (and its **S'** counterpart) have been left out of our optimizations. Even in the restricted form (15) the abstract of an unreduced term such as $\mathbf{SSS}x(\mathbf{SSS})$ can be shorter than that of a reduced form $\mathbf{S}x(\mathbf{S}x)(\mathbf{S'SIS})$. The most common cases where a term involving **S** (or **S'**) can be simplified have been included.

The λ^E algorithm is not always guaranteed to give a shorter abstract than λ^T, especially when **S** is involved. For example,

$$\lambda^T x.\mathbf{S}(\mathbf{K}x)(\mathbf{K}x) \equiv \mathbf{S'SKK},$$

while

$$\lambda^E x.\mathbf{S}(\mathbf{K}x)(\mathbf{K}x) \equiv \mathbf{BK}(\mathbf{SII}).$$

We can overcome any such case by adding an appropriate optimization, in this case,

$$\mathbf{BK}(\mathbf{SII}) \to \mathbf{S'SKK},$$

to our list.

On the other hand it may be that checking for all these optimizations is itself inefficient. In fact any or all of (25)–(42) and even (1), (2), (4) and (16)–(19) can be left out without affecting the results proved below.

Optimizations (1) and (2) can in any case be left out as they can be derived from (16) with (25) and (26). There is a choice between a simpler algorithm and a faster one.

If a term X has no subterms of the form of a left-hand side of any of our optimizations, we will say that it is *fully reduced* (f.r.).

In the examples we have considered the E-algorithm always produced an f.r. combinatory term. It is possible for the algorithm to produce a term that is not f.r., for example:

$$\mathbf{S}(\mathbf{K}X)(\lambda^E x.x) \to \mathbf{B}X(\lambda^E x.x) \equiv \mathbf{B}X\mathbf{I},$$

but, by (25),

$$\mathbf{B} \times \mathbf{I} \to X.$$

We will show that in important cases this does not occur. For combinatory terms and λ-terms, in fact, the E-algorithm provides unique combinatory terms which are optimal in the sense that no other E-optimizations can be applied to them.

LEMMA 1. *The E-optimizations applied to a λ-combinatory term determine a unique f.r. λ-combinatory term.*

PROOF. The E-optimizations can be applied only in the order given, so the only way in which a given term can be optimized in more than one way occurs when one optimization can be applied to two or more parts of a term.

If, for example, a part of a term is of the form $\mathbf{B}X(\mathbf{K}Y)$ and another part $\mathbf{B}X_1(\mathbf{K}Y_1)$, either these parts are disjoint, or the second may occur inside X or Y or the first inside X_1 or Y_1. In either case the result of applying (25) twice is independent of the order.

If part of a term is $\mathbf{SK}X$ and another $\mathbf{SK}X_1$, the same applies unless $\mathbf{SK}X_1$ is part of X. In that case the result of applying (29) to $\mathbf{SK}X$ removes the need for applying it to $\mathbf{SK}X_1$. Either way the final result is $\mathbf{I}$.

A check of all other optimizations shows that in each case a unique result is obtained irrespective of the order in which the optimization is applied in various parts of the term.

The result of the optimization process, as each optimization reduces the length of the term, is a unique f.r. λ-combinatory term.

THEOREM 1. *If Y is a combinatory term then $\lambda^E x_1 \cdots x_n.Y$ is a unique f.r. combinatory term.*

PROOF. Lemma 1 shows that the result of any optimization within Y produces a unique f.r. term.

It is clear then that the lemma follows if we show that, if a combinatory term X is f.r. then $\lambda^E x.X$ is f.r. and unique. We prove this by induction on the length of X.

(i) If $X \equiv x$ then $\lambda^E x.X \equiv \mathbf{I}$, which is f.r.

(ii) If $X \equiv a$, where a is a primitive term other than x, then $\lambda^E x.X \equiv \mathbf{K}a$, which is f.r.

We now assume that the theorem holds for terms shorter than X.

(iii) If $X \equiv xY$ where $x \notin \mathrm{FV}(Y)$, then $\lambda^E x.X \equiv \mathbf{CI}Y$, which is f.r. (the only left-hand sides of optimizations starting with $\mathbf{C}$ are $\mathbf{C}(\mathbf{B}UV)$ and $\mathbf{C}(\mathbf{K}U)V$).

(iv) If $X \equiv xY$ where $x \in \mathrm{FV}(Y)$, then $\lambda^E x.X \equiv \mathbf{SI}(\lambda^E x.Y)$, where by the inductive hypothesis $\lambda^E x.Y$ is f.r. This can only include the left-hand side of an optimization (4) if $\lambda^E x.Y \equiv \mathbf{K}Z$ for some Z. As $x \in \mathrm{FV}(Y)$, Y can only be $\mathbf{K}Zx$, but then X is not f.r. Thus $\lambda^E x.X$ must be f.r.

(v) If $X \equiv Yx$ where $x \notin \mathrm{FV}(Y)$, then $\lambda^E x.X \equiv Y$, which is f.r. as X is f.r.

(vi) If $X \equiv aY$, where a is a primitive other than x, $Y \not\equiv x$ and $x \in \mathrm{FV}(Y)$, then $\lambda^E x.X \equiv \mathbf{B}a(\lambda^E x.Y)$. This can be a left-hand side of an optimization if $\lambda^E x.Y \equiv \mathbf{I}$ or $\mathbf{K}Z$ for some Z or if $\mathbf{I}$ is primitive and $a \equiv \mathbf{I}$. The first case is impossible as $\mathbf{Y} \not\equiv x$, in the second case Y would have to be $\mathbf{K}Zx$ and X would not be f.r., and in the third case $X \equiv \mathbf{I}Y$ would also not be f.r. Thus $\lambda^E x.X$ must be f.r.

(vii) If $X \equiv UVY$ where $x \in \mathrm{FV}(Y) - \mathrm{FV}(UV)$, then $\lambda^E x.X \equiv \mathbf{B}'UV(\lambda^E x.Y)$. This can only fail to be f.r. if U is $\mathbf{I}$ or $\mathbf{K}$; in those cases X would not be f.r.

(viii) If $x \notin \mathrm{FV}(X)$, then $\lambda^E x.X \equiv \mathbf{K}X$, which is f.r..

(ix) If $X \equiv UVY$ where $x \in \mathrm{FV}(V) \cap \mathrm{FV}(Y) - \mathrm{FV}(U)$, then

$$\lambda^E x.X \equiv \mathbf{S}'U(\lambda^E x.V)(\lambda^E x.Y).$$

This can fail to be f.r. only if $U \equiv \mathbf{K}$, $\mathbf{I}$ or $\mathbf{K}Z$ (in these cases X is not f.r.) or if $\lambda^E x.V$ or $\lambda^E x.Y \equiv \mathbf{K}Z$, which is impossible as above.

(x) If $X \equiv UVY$ where $x \in \mathrm{FV}(V) - \mathrm{FV}(UY)$, then $\lambda^E x.X \equiv \mathbf{C}'U(\lambda^E x.V)Y$. This is f.r. as U cannot be $\mathbf{I}$ or $\mathbf{K}$.

(xi) If $X \equiv UVY$ where $x \in \mathrm{FV}(V) \cap \mathrm{FV}(U) - \mathrm{FV}(Y)$, then $\lambda^E x.X \equiv \mathbf{C}(\lambda^E x.UV)Y$. This is f.r. as $\lambda^E x.UV$ will start with $\mathbf{S}$ or $\mathbf{S}'$.

(xii) If $X \equiv UVY$ and $x \in \mathrm{FV}(U) \cap \mathrm{FV}(V) \cap \mathrm{FV}(Y)$, then $\lambda^E x.UV$ must start with $\mathbf{S}$ or $\mathbf{S}'$ and $\lambda^E x.Y$ cannot be $\mathbf{K}Z$.

(xiii) If $X \equiv UVY$ and $x \in \mathrm{FV}(U) \cap \mathrm{FV}(Y) - \mathrm{FV}(V)$, then

$$\lambda^E x.X \equiv \mathbf{S}(\lambda^E x.UV)(\lambda^E x.Y).$$

This is f.r. as $\lambda^E x.UV$ must start with $\mathbf{C}$ or $\mathbf{C}'$ and $\lambda^E x.Y \equiv \mathbf{K}Z$.

(xiv) If $X \equiv UVY$ and $x \in \mathrm{FV}(U) - \mathrm{FV}(VY)$, then $\lambda^E x.X \equiv \mathbf{C}(\lambda^E x.UV)Y$, where $\lambda^E x.UV$ starts with $\mathbf{C}$ or $\mathbf{C}'$. Again X must be f.r.

This completes the proof of the theorem.

THEOREM 2. *The E-algorithm applied to a λ-term produces a unique f.r. combinatory term. (Note the E-algorithm, in this case, consists of the $\beta\eta$-optimizations followed by applications of the* (abcd′e′f′def) *algorithm.)*

PROOF. By Lemma 1 the E- (i.e. $\beta\eta$-) optimizations produce a unique f.r. (λ-) term.

It is clear that no abstraction step applied within an f.r. λ-term can produce a $\beta\eta$-redex. We will show that also no other left-hand side of an E-optimization rule can be produced.

We note first that if $\mathbf{S}UV$ is produced by an abstraction step then $\mathbf{S}UV \equiv \lambda^E x.PQ$ where $\lambda^E x.P \equiv U$ and $\lambda^E x.Q \equiv V$. (Note that the possibility of $\mathbf{S}UV$ coming from η abstraction steps applied to $\mathbf{S}UVy_1 \cdots y_n$ does not arise as $\mathbf{S}UVy_1 \cdots y_n$ can only arise if $\mathbf{S}UV$ comes from an abstraction step; $y_1 \cdots y_n$ would then previously have been removed by the optimization (22).) Similarly $\mathbf{K}U$ is produced by an abstraction only if $\mathbf{K}U \equiv \lambda^E x.U$ and $x \notin \mathrm{FV}(U)$ and $\mathbf{I}$ is produced only by $\mathbf{I} \equiv \lambda^E x.x$. Similarly for terms of the forms $\mathbf{B}UV$, $\mathbf{C}UV$, $\mathbf{B}'UVT$, $\mathbf{C}'UVT$, $\mathbf{S}'UV$.

It then follows that $\mathbf{S}(\mathbf{K}U)(\mathbf{K}V)$ can come only from $\lambda^E x.Xx$ where $x \notin \mathrm{FV}(X)$, which is impossible as $\lambda^E x.Xx = X$.

Similarly we can exclude all the left-hand sides of all other E-optimizations.

Note that additional optimizations such as

$$\mathbf{C}'X\mathbf{I} \to \mathbf{C}X \quad \text{and} \quad \mathbf{BK}(\mathbf{SII}) \to \mathbf{S}'\mathbf{SKK}$$

can be added requiring at most minor changes to the proof of Theorems 1 and 2.

It is important to note that it is not decidable whether two arbitrary combinatory terms are equal (even in the weak sense), so there can be no simple general algorithm for finding a shortest abstract of a term.

Extensions of some other algorithms. The Schönfinkel (abcdef) algorithm can be extended exactly as above by applying the optimizations (11)–(14), (1), (2), (4), (16) and (25)–(31) before abstracting. The Curry (abf) algorithm can be extended by applying (11), (12), (1), (2) and (29) before abstracting. Counterparts to Theorems 1 and 2 can easily be proved.

Algorithms involving supercombinators such as that of Abdali can also be extended and the abstracts simplified if optimizations (11), (12), (1), (2) and (29) are carried out before any abstraction. (As well as (4), (16), (30), (31) and (25)–(28) if **B** and **C** are included among the primitives.)

Abdali's generalised combinators are $\mathbf{K}_n$, $\mathbf{I}_n^m$ and $\mathbf{B}_n^m$. They have the properties:

$$\begin{aligned}\mathbf{K}_n X Y_1 \cdots Y_n &= X,\\ \mathbf{I}_n^m Y_1 \cdots Y_n &= Y_m,\\ \mathbf{B}_n^m X Y_1 \cdots Y_m Z_1 \cdots Z_n &= X(Y_1 Z_1 \cdots Z_n)\cdots(Y_m Z_1 \cdots Z_n).\end{aligned}$$

We now list some optimizations involving $\mathbf{K}_n$, $\mathbf{I}_n^m$ and $\mathbf{B}_n^m$, that could precede the abstraction process outlined in [1]:

(43) $$\mathbf{K}_n X Y_1 \cdots Y_n \to X,$$

(44) $$\mathbf{I}_n^m Y_1 \cdots Y_n \to Y_m,$$

$$\mathbf{K}_n X Y_1 \cdots Y_k \to \mathbf{K}_{n-k} X, \qquad (k < n)$$

$$\mathbf{B}_n^{m+1} K_m \to \mathbf{K}_{n+m},$$

$$\mathbf{I}_n^m Y_1 \cdots Y_k \to \mathbf{K}_{n-k} Y_m, \qquad (m \le k < n)$$

$$\mathbf{I}_n^m Y_1 \cdots Y_k \to \mathbf{I}_{n-k}^{m-k}, \qquad (k < m \le n)$$

$$\mathbf{B}_k^n \mathbf{I}_n^m \to \mathbf{I}_n^m, \qquad (1 \le m < n)$$

$$\mathbf{B}_n^m \mathbf{K}_k Y_1 \cdots Y_{k+1} \to \mathbf{B}_n^{m-k} \mathbf{I} Y_1, \qquad (1 \le k < m)$$

$$\mathbf{B}_n^m (\mathbf{K}_k X) Y_1 \cdots Y_k \to \mathbf{B}_n^{m-k} X, \qquad (1 \le k \le m)$$

$$\mathbf{B}_n^m X \mathbf{K}_k \mathbf{K}_k \cdots \mathbf{K}_k Z_1 \cdots Z_{k+1} \to \mathbf{B}_{n-k}^m X Z_1 \cdots Z_1 \qquad (1 \le k < n)$$

(where there are m $\mathbf{K}_k$'s on the left and m Z_1's on the right),

$$\mathbf{B}_n^m X(\mathbf{K}_k Y_1)\cdots(\mathbf{K}_k Y_m) Z_1 \cdots Z_k \to \mathbf{B}_m^{n-k} X Y_1 \cdots Y_m, \qquad (1 \le k < n)$$

$$\mathbf{B}_n^m X \mathbf{I}_1^n \mathbf{I}_2^n \cdots \mathbf{I}_k^n \to X.$$

(These were obtained by abstracting both sides of (43) and (44) or substituting into the equation for $\mathbf{B}_n^m$.)

Theorems similar to Theorems 1 and 2 can again be proved.

The Burton improvements to Turner's algorithm. Burton in [4] has suggested a change to Turner's algorithm which, he claims, for a certain class of terms, reduces the length of the combinatory term produced by the algorithm to $O(n)$, where n is the number of atoms in the original λ-combinatory term. According to Kennaway in [9] however, no more than $O(n \log n)$ should be claimed, which is also available by the Kennaway and Sleep director string method in [11].

While this latter method is also applicable to the E-algorithm, that of Burton runs rather counter to ours in that it introduces new β-redexes into a term. Kennaway also regards this as a drawback.

Let us compare, using our earlier example, the Turner algorithm, the Burton algorithm, the E-algorithm and a combined Burton-E-algorithm.

$$\lambda^{\mathrm{T}} x_4.x_1(x_2(x_3x_4)) = \mathbf{B}x_1(\mathbf{B}x_2(\mathbf{B}x_3\mathbf{I})),$$

after 7 abstraction operations and 3 optimizations.

The Burton balancing operation requires instead the evaluation of

$$\lambda^{\mathrm{T}} x_4.(\lambda v.x_1(x_2v))(x_3x_4) = \mathbf{B}(\mathbf{B}x_1x_2)(\mathbf{B}x_3\mathbf{I}).$$

This requires 4 abstraction operations and 2 optimizations for the inner λ-term. This inner term is then treated as an atom resulting in the need of only 4 more abstractions and 2 more optimizations. This algorithm is faster than Turner's because the inner and outer abstractions can be carried out independently.

$$\lambda^{E} x_4.x_1(x_2(x_3x_4)) = \mathbf{B}x_1(\mathbf{B}x_2x_3).$$

Here E is exactly (abcd′e′f′def) and so requires only 3 abstractions and no optimizations.

If we balance first and do this using a combinator rather than a λ-term we have

$$\lambda^{E} x_4.\mathbf{B}x_1x_2(x_3x_4) = \mathbf{B}(\mathbf{B}x_1x_2)x_3.$$

This required (after the balancing) only 2 abstractions.

Note that if we wish to evaluate $\lambda^* x_1x_2x_3x_4.x_1(x_2(x_3x_4))$, the Turner algorithm requires altogether 46 λ-abstractions and 27 optimizations to give an abstract of 10 combinators. The Burton algorithm requires 50 λ-abstractions and 25 optimizations to give a (different but $\beta\eta$-equal) abstract of 10 combinators. The E algorithm requires 9 λ-abstractions and no optimizations to give $\mathbf{C}'(\mathbf{B}'(\mathbf{B}'\mathbf{B}))\mathbf{I}\mathbf{B}$. The E-algorithm applied after balancing using the combinator $\mathbf{B}$ requires 7 λ-abstractions and no optimizations, and produces the abstract $\mathbf{B}'\mathbf{B}\mathbf{B}\mathbf{B}$.

This example clearly shows the advantage of the E-algorithm over the Turner and Burton algorithms, but it also shows that there are optimizations not included in the E-algorithm that could simplify abstracts even further.

The use of balancing before applying the E-algorithm, however, does not always improve efficiency nor the simplicity of the outcome. For example

$$\lambda^{E} x_1x_2x_3.x_2(x_1(x_2x_3)) = \mathbf{B}(\mathbf{S}\mathbf{B})\mathbf{B}$$

in 8 λ-abstractions, while

$$\lambda^{E} x_1x_2x_3.\mathbf{B}x_2x_1(x_2x_3) = \mathbf{C}(\mathbf{B}\mathbf{S}(\mathbf{C}\mathbf{B}))\mathbf{I}$$

after the balancing step and 9 λ-abstractions.

It is probably not possible to formulate an algorithm where all the optimizations are performed before the first abstraction and which (perhaps using combinator balancing) always produces the simplest possible abstract. There are probably always further complex optimizations such as

$$\mathbf{C(B'(B'B))IB} \to \mathbf{B'BBB},$$

discovered in the first example above, which can be added to improve (and complicate) the algorithm.

Conclusion. We have shown that the Turner algorithm, which is considered as the best of the current abstraction algorithms, in terms of simplicity and efficiency, does not produce an abstract $Y = [x]X$ which is such that $Yx > X$, but a Y such that $Yx =_{\beta\eta} X$.

The extended algorithm introduced in this paper, obtained by performing optimizations before rather than after abstractions and using the (abcd'e'f'def) algorithm, is much more efficient still and, in general, produces simpler abstracts. This algorithm can sometimes be improved further by the use of Burton's balancing, but a detailed analysis would have to be made as to when balancing is needed.

The optimization technique, here applied to Turner's algorithm, can also be applied to others.

REFERENCES

[1] S. K. ABDALI, *An abstraction algorithm for combinatory logic*, this JOURNAL, vol. 41 (1976), pp. 222–224.

[2] M. W. BUNDER, *On adding further forms of* (ξ) *to weak equality in combinatory logic*, Preprint No. 8/87, Department of Mathematics, University of Wollongong, Wollongong, 1987.

[3] M. W. BUNDER, J. R. HINDLEY and J. P. SELDIN, *On adding* (ξ) *to weak equality*, this JOURNAL, vol. 54 (1989), pp. 590–607.

[4] F. W. BURTON, *A linear space translation of functional programs to Turner combinators*, ***Information Processing Letters***, vol. 14 (1982), pp. 201–204.

[5] P. L. CURIEN, ***Categorical combinators, sequential algorithms and functional programming***, Research Notes in Theoretical Computer Science, Pitman, London, and Wiley, New York, 1986.

[6] H. B. CURRY and R. FEYS, ***Combinatory logic***. Vol. 1, North-Holland, Amsterdam, 1958.

[7] R. J. M. HUGHES, ***The design and implementation of programming languages***, Technical Monograph PR6-40, Oxford University Computing Laboratory, Oxford, 1983.

[8] ———, *Super-combinators*, ***Conference record of the 1982 ACM symposium on LISP and functional programming***, pp. 1–10.

[9] J. R. KENNAWAY, *The complexity of a translation of* λ*-calculus to combinators*, Internal Report CSA/13/1984, Declarative Systems Architecture Group, University of East Anglia, Norwich, 1984.

[10] J. R. KENNAWAY and M. R. SLEEP, *Counting director strings*, Preprint, University of East Anglia, Norwich, 1984.

[11] ———, *Variable abstraction in* $O(n \log n)$ *space*, ***Information Processing Letters***, vol. 24 (1987), pp. 343–349.

[12] V. KRISHNAMURTHY, ***Parallelism in functional languages using combinators and delayed evaluation***, Ph.D. thesis, University of Waikato, Hamilton, New Zealand, 1986.

[13] R. D. LINS, *Categorical multi-combinators*, ***Functional programming languages and computer architecture*** (G. Kahn, editor), Lecture Notes in Computer Science, vol. 274, Springer-Verlag, Berlin, 1987, pp. 60–79.

[14] J. C. MULDER, *Complexity of combinatorial code*, Preprint No. 389, Department of Mathematics, University of Utrecht, Utrecht, 1985.

[15] H. G. OBERHAUSER, *On the correspondence of lambda style reduction and combinator style reduction*, ***Graph reduction (proceedings, Santa Fe, New Mexico, 1986)***, Lecture Notes in Computer Science, vol. 279, Springer-Verlag, Berlin, 1987, pp. 1–25.

[16] S. L. PEYTON JONES ***The implementation of functional programming languages***, Prentice-Hall, Englewood Cliffs, New Jersey, 1986.

[17] M. SCHÖNFINKEL, *Über die Bausteine der mathematischen Logik*, ***Mathematische Annalen***, vol. 92 (1924), pp. 305–316; English translation, ***From Frege to Gödel: a source-book in mathematical logic*** (J. van Heijenoort, editor), Harvard University Press, Cambridge, Massachusetts, 1967, pp. 355–366.

[18] R. STATMAN, *An optimal translation of λ terms into combinators*, preprint, Carnegie-Mellon University, Pittsburgh, Pennsylvania, 1983.

[19] M. M. E. TONINO, private communication, 1984 (see Mulder [14]).

[20] D. A. TURNER, *Another algorithm for bracket abstraction*, this JOURNAL, vol. 44 (1979), pp. 267–270.

FACULTY OF MATHEMATICAL SCIENCES
UNIVERSITY OF WOLLONGONG
WOLLONGONG, NEW SOUTH WALES 2500, AUSTRALIA