Comput. Lang. Vol. 10, No. 3/4, pp. 211–224, 1985
Printed in Great Britain.

0096-0551/85 $3.00 + 0.00

EFFICIENT COMBINATOR CODE*

M. S. JOY†, V. J. RAYWARD-SMITH and F. W. BURTON‡
School of Computing Studies and Accountancy, University of East Anglia, Norwich NR4 7TJ, U.K.

(*Received* 26 *September* 1984; *revision received* 6 *March* 1985)

Abstract—Some combinatory logics are examined as object code for functional programs. The worst-case performances of certain algorithms for abstracting variables from combinatory expressions are analysed. A lower bound on the performance of any abstraction algorithm for a finite set of combinators is given. Using the combinators S, K, I, B, C, S′, B′, C′ and Y, the problem of finding an optimal abstraction algorithm is shown to be NP-complete. Some methods of improving abstraction algorithms for those combinators are examined, including "balancing" (for asymptotic performance) and "peephole" optimisations (for smaller cases).

Combinators Combinatory logic Computational complexity Functional languages NP-completeness Optimisation

1. INTRODUCTION

Combinatory logic [1–3] had for many years been of interest to just a handful of logicians. However, it is now recognised that combinators may be a useful low-level code for functional programs. This idea has stimulated several papers, notably by Turner [4, 5] and there exist machines which use combinators as such a code (such as SKIM [6]). We examine here some aspects of efficient translation of a functional program to combinator code.

We assume the reader is familiar with the rudiments of combinatory logic (an excellent introduction can be found in Ref. [1]; Ref. [2] is a comprehensive treatment of the subject and a discussion of recent trends can be found in Ref. [3]). The material in this paper is a summary of some of the results obtained in Ref. [7]. Detailed proofs of the results will be omitted.

A combinatory logic is defined by an alphabet consisting of the symbols "(" and ")", together with a set of variables, a set of annotated constants, and a set of combinators. An atom is a variable or a constant or a combinator. A combinatory expression E is either x (where x is an atom) or (F G) (where F and G are combinatory expressions). If E is a combinatory expression, then size (E) (or |E|) will mean the number of atoms occurring in E. The symbol " ≡ " will stand for lexical equality. To each combinator is associated a reduction rule which is used to change one combinatory expression into another. If just one such change is made that change is called a reduction step, and if a sequence of changes are made then that is called a reduction. The logics will have names beginning "CL-" and we now give the combinators and combinatory logics we shall use.

In the following definitions of the reduction rules the symbols "a", "b", "c", "d" are to be regarded as arbitrary combinatory expressions.

S a b c	>	(a c) (b c)	B′ d a b c	>	d a (b c)
K a b	>	a	C′ d a b c	>	d (a c) b
I a	>	a	J a b c	>	a b
B a b c	>	a (b c)	J′ d a b c	>	d a b
C a b c	>	a c b	Y a	>	a (Y a)
S′ d a b c	>	d (a c) (b c)			

*This material is based on work supported by the United Kingdom Science and Engineering Research Council and by the National Science Foundation under grant number ECS-8312748.
†Present address: School of Computing and Information Technology, The Polytechnic, Wolverhampton WV1 1LY, U.K.
‡Present address: Department of Electrical Engineering and Computer Science, University of Colorado at Denver, 1100 Fourteenth Street, Denver, CO 80202, U.S.A.

Name of Logic	Set of Combinators
CL-SKI	{Y, S, K, I}
CL-SKIBC	{Y, S, K, I, B, C}
CL-Dash	{Y, S, K, I, B, C, S′, B′, C′}
CL-J	{Y, S, B, C, J, S′, B′, C′, J′, I}

We shall assume the expressions are represented as directed acyclic graphs, so that expressions may be shared. An expression which can have no more reductions applied to it is in normal form. The functions oc (occurs in), and its converse /oc (does not occur in), are defined in the obvious way. The symbol " = " used between combinatory expressions will indicate that they can be interconverted by using a number of reduction steps or expansions (the inverse of reductions). If E, F and G are combinatory expressions, then [E/F]G shall mean the expression constructed by replacing all occurrences of the expression F in G by E.

An abstraction algorithm is a function taking as argument a combinatory expression containing variables and returning an expression containing fewer variables such that, if the variables thus removed are appended to the resulting expression, and that expression reduced fully to normal form, (the normal form of) the original expression will be arrived at again. Abstraction algorithms will be given names of the form Abs/'logic-name'/'number', where 'logic-name' refers to the name of the combinatory logic for which the algorithm is designed, and 'number' the number of the algorithm (for each logic we shall consider several algorithms).

We shall write abstraction algorithms as functions of two arguments, the first a list of the variables being abstracted. Each algorithm in the next section abstracts variables "one at a time". So we shall in each case define the algorithm for abstracting one variable only, and the following rule is to be assumed:

$$\text{Abs/X/r}(x_1, \ldots, x_m)(E) \equiv \text{Abs/X/r}(x_1, \ldots, x_{m-1})((\text{Abs/X/r}(x_m)(E))),$$

where X is the logic-name, and r is the algorithm number.

Standard notations for the complexity of functions—O(n), $\theta(n)$ and $\Omega(n)$—will be used (definitions can be found in Ref. [9]).

2. ABSTRACTION ALGORITHMS

In this section we define some abstraction algorithms and give worst-case performance bounds.

Definition: Abs/SKI/1

The first of the following that is applicable should be used:

$$\text{Abs/SKI/1(x)(x)} \equiv \text{I}, \tag{i}$$

$$\text{Abs/SKI/1(x)(y)} \equiv \text{(K y) if atom(y) and } y \neq x, \tag{ii}$$

$$\text{Abs/SKI/1(x)(E F)} \equiv \text{(S(Abs/SKI/1(x)(E))(Abs/SKI/1(x)(F}))). \tag{iii}$$

This is arguably the "simplest" abstraction algorithm, being easy to state and easy to analyse—unfortunately, it does not produce compact combinator code. In an attempt to improve this situation, the following two variations have been proposed.

Definition: Abs/SKI/2

As Abs/SKI/1, except (ii) is replaced by:

$$\text{Abs/SKI/2(x)(E)} \equiv \text{(K E) if x /oc E.} \tag{iv}$$

Definition: Abs/SKI/3

As for Abs/SKI/2, except that after (iv) is inserted:

$$\text{Abs/SKI/3(x)(E x)} \equiv \text{E if x /oc E.} \tag{v}$$

Abs/SKI/3 includes "eta-abstraction", equivalent to "strong" (as opposed to "weak") combinatory logic. That is, two functions are equated if they evaluate to the same result given the same arguments. Eta-abstraction makes the underlying theory somewhat more complicated, but when variables are abstracted using algorithms that allow eta-abstraction, then for many expressions the size of the resulting code is significantly reduced. This is especially true for small expressions corresponding to functions that might appear in a "real" program.

Definition: Abs/SKI/4

This algorithm is equivalent to Abs/SKI/3, but is defined in terms of rewrite rules (see Ref. [4]) rather than "occurs in".

$$\begin{aligned}
&\text{Abs/SKI/4(x)(x)} &&\equiv \text{I},\\
&\text{Abs/SKI/4(x)(y)} &&\equiv \text{(K y) if atom(y) and } y \neq x,\\
&\text{Abs/SKI/4(x)(E F)} &&\equiv \text{S(Abs/SKI/4(x)(E))(Abs/SKI/4(x)(F))},
\end{aligned}$$

but, whenever a term of the form $S(K\ E_1)(K\ E_2)$ occurs, replace it immediately by $K(E_1\ E_2)$, and whenever a term of the form $S(K\ E_3)I$ occurs, replace it immediately by E_3.

These we will refer to as "optimisations" for the rest of this section, and in future algorithms will use the shorthand

$$S(K\ E_1)(K\ E_2) \rightarrow K(E_1\ E_2), \tag{vi}$$

$$S(K\ E_3)I \rightarrow E_3, \tag{vii}$$

etc.

Theorem

Let E be an expression in CL-SKI, then

$$\text{Abs/SKI/3}(x_1, \ldots, x_m)(E) \equiv \text{Abs/SKI/4}(x_1, \ldots, x_m)(E).$$

By introducing the combinators B and C we are able to make a substantial improvement—CL-SKIBC avoids the excessive verbosity of CL-SKI, although the asymptotic performance is similar. These two combinators were initially studied as computer code in Ref. [4].

Definition: Abs/SKIBC/1

$$\text{Abs/SKIBC/1(x)(x)} \equiv \text{I}, \tag{viii}$$

$$\text{Abs/SKIBC/1(x)(E)} \equiv \text{(K E) if x /oc E}, \tag{ix}$$

$$\text{Abs/SKIBC/1(x)(E F)} \equiv \text{(S(Abs/SKIBC/1(x)(E))(Abs/SKIBC/1(x)(F))) if x oc E and x oc F}, \tag{x}$$

$$\text{Abs/SKIBC/1(x)(E F)} \equiv \text{(C(Abs/SKIBC/1(x)(E))F) if x oc E and x /oc F}, \tag{xi}$$

$$\text{Abs/SKIBC/1(x)(E F)} \equiv \text{(B E (Abs/SKIBC/1(x)(F))) if x /oc E and x oc F}. \tag{xii}$$

The following two variations have also been proposed. Abs/SKIBC/2 contains eta-abstraction, Abs/SKIBC/3 uses rewrite rules and was introduced in Ref. [4].

Definition: Abs/SKIBC/2

As for Abs/SKIBC/1, except after (vii) is inserted:

$$\text{Abs/SKIBC/2(x)(E x)} \equiv \text{E, if x /oc E}. \tag{xiii}$$

Definition: Abs/SKIBC/3

As Abs/SKIBC/4, but with additional optimisations:

$$S\ (K\ E_1)\ E_2 \rightarrow B\ E_1\ E_2, \tag{xiv}$$

$$S\ E_1\ (K\ E_2) \rightarrow C\ E_1\ E_2. \tag{xv}$$

Theorem

Let E be an expression in CL-SKIBC, then

$$\text{Abs/SKIBC/2}(x_1, \ldots, x_m)\,(E) \equiv \text{Abs/SKIBC/3}(x_1, \ldots, x_m)\,(E).$$

By introducing the "dashed" or "long-reach" combinators we can make an improvement in the asymptotic performance of the simple abstraction algorithms. The extra combinators in CL-Dash were initially examined in Ref. [5], where the algorithm Abs/Dash/3 is proposed.

Definition: Abs/Dash/1

The first of the following which is applicable should be used (k is an arbitrary expression containing no variables):

$$\text{Abs/Dash/1}(x)(x) \equiv I, \tag{xvi}$$

$$\text{Abs/Dash/1}(x)(E) \equiv (K\ E) \text{ if } x \text{ /oc } E, \tag{xvii}$$

$$\text{Abs/Dash/1}(x)(k\ E\ F) \equiv (S'\ k\ (\text{Abs/Dash/1}(x)(E))\ (\text{Abs/Dash/1}(x)(F))) \text{ if } x \text{ oc } E \text{ and } x \text{ oc } F, \tag{xviii}$$

$$\text{Abs/Dash/1}(x)(k\ E\ F) \equiv (C'\ k\ (\text{Abs/Dash/1}(x)(E))F) \text{ if } x \text{ oc } E \text{ and } x \text{ /oc } F, \tag{xix}$$

$$\text{Abs/Dash/1}(x)(k\ E\ F) \equiv (B'\ k\ E\ (\text{Abs/Dash/1}(x)(F))) \text{ if } x \text{ /oc } E \text{ and } x \text{ oc } F, \tag{xx}$$

$$\text{Abs/Dash/1}(x)(E\ F) \equiv (S\ (\text{Abs/Dash/1}(x)(E))\ (\text{Abs/Dash/1}(x)(F))) \text{ if } x \text{ oc } E \text{ and } x \text{ oc } F \text{ and } E \neq (k\ E_1) \text{ for any } E_1, \tag{xxi}$$

$$\text{Abs/Dash/1}(x)(E\ F) \equiv (C\ (\text{Abs/Dash/1}(x)(E))F) \text{ if } x \text{ oc } E \text{ and } x \text{ /oc } F \text{ and } E \neq (k\ E_1) \text{ for any } E_1, \tag{xxii}$$

$$\text{Abs/Dash/1}(x)(E\ F) \equiv (B\ E\ (\text{Abs/Dash/1}(x)(F))) \text{ if } x \text{ /oc } E \text{ and } x \text{ oc } F \text{ and } E \neq (k\ E_1) \text{ for any } E_1. \tag{xxiii}$$

Definition: Abs/Dash/2

As for Abs/Dash/1, but after (xxi) is inserted

$$\text{Abs/Dash/2}(x)(E\ x) \equiv E \text{ if } x \text{ /oc } E. \tag{xxiv}$$

Theorem

Let E be an expression in CL-Dash, then

$$|\text{Abs/Dash/2}(x_1, \ldots, x_m)(E)| \leq |\text{Abs/Dash/1}(x_1, \ldots, x_m)(E)|.$$

Definition: Abs/Dash/3

As Abs/SKIBC/3, but with the following extra optimisations:

$$S(B\ E_1\ E_2)\ E_3 \rightarrow S'\ E_1\ E_2\ E_3, \tag{xxv}$$

$$B(E_1\ E_2)\ E_3 \rightarrow B'\ E_1\ E_2\ E_3, \tag{xxvi}$$

$$C(B\ E_1\ E_2)\ E_3 \rightarrow C'\ E_1\ E_2\ E_3. \tag{xxvii}$$

We note that the order of the optimisations matters. For instance, consider the function $(\lambda x.\ +\ x\ x)$. Using Abs/Dash/3 we get

$$\text{Abs/Dash/3}(x)(+\ x\ x)$$

$$\rightarrow S(\text{Abs/Dash/3}(x)(+\ x))(\text{Abs/Dash/3}(x)(x)) \quad \text{(using iii)}$$

$$\rightarrow S(S(\text{Abs/Dash/3}(x)(+))(\text{Abs/Dash/3}(x)(x)))I \quad \text{(using iii and i)}$$

$$\rightarrow S(S(K+)I)I \quad \text{(using iv and i)}$$

$$\rightarrow S\ +\ I. \quad \text{(using vii)}$$

However, using a different order, we get

Abs/Dash/3(x)(+ x x)

→ . . . → S(S(K +)I)I

→ S(B + I)I (using xiv)

→ S′ + I I. (using xxv)

The algorithms we give perform these optimisations in the most efficient order, and yield a unique result, provided one always uses the first one which can be used.

By extending the sets of optimisations (which remain finite) in each of the above algorithms, we can dispense with the necessity of performing their constituent operations in any particular order. Thus such "extended" abstraction algorithms possess the Church–Rosser "confluence" property. We do not exhibit them here, since the resulting abstraction algorithms are more cumbersome than those we use whilst still yielding the same results.

Theorem

Let E be an expression in CL-Dash, then

$$\text{Abs/Dash/2}(x_1, \ldots, x_m)(E) \equiv \text{Abs/Dash/3}(x_1, \ldots, x_m)(E).$$

CL-J is very similar to CL-Dash, except the combinator K is replaced by two combinators—J and J′—whose degrees (number of arguments they need before they can reduce) correspond to S, B and C (for J) and S′, B′ and C′ (for J′). This fits in with the idea that combinators can "tag" the internal nodes of a graph representing a combinator expression and "ship down" arguments to left or right (or neither or both) immediate subexpression of that node. A different formalisation of this concept is "Director Strings" [8]. The two abstraction algorithms we now present correspond to Abs/Dash/1 and Abs/Dash/2.

Definition: Abs/J/1

The first of the following which is applicable should be used (k is an arbitrary expression containing no variables):

Abs/J/1(x)(x) ≡ I, (xxviii)

Abs/J/1(x)(E) ≡ (J I E) if x /oc E and atom(E), (xxix)

Abs/J/1(x)(k E F) ≡ (S′ k (Abs/J/1(x)(E)) (Abs/J/1(x)(F)))
if x oc E and x oc F, (xxx)

Abs/J/1(x)(k E F) ≡ (C′ k (Abs/J/1(x)(E))F)
if x oc E and x /oc F, (xxxi)

Abs/J/1(x)(k E F) ≡ (B′ k E (Abs/J/1(x)(F)))
if x /oc E and x oc F, (xxxii)

Abs/J/1(x)(k E F) ≡ (J′ k E F)
if x /oc E and x /oc F, (xxxiii)

Abs/J/1(x)(E F) ≡ (S (Abs/J/1(x)(E)) (Abs/J/1(x)(F)))
if x oc E and x oc F and E ≠ (k E_1) for any E_1, (xxxiv)

Abs/J/1(x)(E F) ≡ (C (Abs/J/1(x)(E)) F)
if x oc E and x /oc F and E ≠ (k E_1) for any E_1, (xxxv)

Abs/J/1(x)(E F) ≡ (B E (Abs/J/1(x)(F)))
if x /oc E and x oc F and E ≠ (k E_1) for any E_1, (xxxvi)

Abs/J/1(x)(E F) ≡ (J E F)
if x /oc E and x /oc F and E ≠ (k E_1) for any E_1. (xxxvii)

Definition: Abs/J/2

As Abs/J/1, except after (xxi) is inserted

$$\text{Abs/J/2}(x)(E\,x) \equiv E \text{ if } x/oc\ E. \qquad \text{(xxxviii)}$$

From Table 1 we see that the abstraction algorithms for CL-Dash and CL-J yield asymptotically better code than those for CL-SKI and CL-SKIBC. We can remedy the asymptotic performance by performing one of the abstraction algorithms in CL-Dash that we have described, and then replacing each dashed combinator in CL-Dash by an equivalent expression in CL-SKIBC. CL-SKI can be treated in a similar manner, except B and C will also need replacing. However, this is a lengthy method of producing asymptotically better code for CL-SKI and CL-SKIBC, and the code so produced will always be worse than that produced using just CL-Dash (typically by a factor of 10).

The logics introduced above form a hierarchy in the sense that, except for CL-J, the successive logics are formed by adding increasing numbers of combinators to the logic which contains only S, K and I. Thus one can consider the sequence of logics as increasing in complexity. As is to be expected, the more combinators are added to the logic, the shorter the code produced when a naïve algorithm is used to abstract variables.

Results of "worst case" analyses for those algorithms are summarised in Table 1. If we are considering the algorithm Abs/X/r, then the column indicated by "upper bound" will contain a provable upper bound on the value of $|\text{Abs/X/r}(x_1, \ldots, x_m)(E)|$, where E ranges over expressions containing variables in CL-X of size n. The column indicated by "attainable" will give the size of code for an achievable example. It will be assumed that $m \leq 3$ and $n - m > 2$. Proofs of all these results are to be found in Ref. [7].

We have shown one algorithm to be "exponential", and several to be either "$\theta(n*m^2)$" or "$\theta(n*m)$". The latter are clearly those which are likely to be of practical use, and as we shall see later, Abs/Dash/2 and Abs/Dash/3 yield particularly good code which is not easily susceptible to optimisation.

Table 1.

Algorithm	Upper bound	Attainable
Abs/SKI/1	$3^m n - (3^m - 1)/2$	$2*m^{m-1}n - (3^m - 1)/2$
Abs/SKI/2	$2nm^2 + 4nm - n$	$n(m^2 + m + 1) - 2m^3/3 + 2m/3 - 2$
Abs/SKI/3	$2nm^2 + 4nm - 4n$	$n(m^2 + m + 1) - 2m^3/3 - 7m/3$
Abs/SKI/4	$2nm^2 + 4nm - 4n$	$n(m^2 + m + 1) - 2m^3/3 - 7m/3$
Abs/SKIBC/1	$n(m + 1/2)^2/2 + n(2m - 1)$	$nm(m + 1) - (2m^3 + 3m^2 + 11m - 6)/6$
Abs/SKIBC/2	$nm^2/2 + 2nm - 21n/8$	$nm(m + 1) - (m^3 + 2m - 3)/3$
Abs/SKIBC/3	$nm^2/2 + 2nm - 21n/8$	$nm(m + 1) - (m^3 + 2m - 3)/3$
Abs/Dash/1	$nm + n - n$	$nm + n - (m^2 - m + 2)/2$
Abs/Dash/2	$nm + n - n$	$nm + n - (m^2 + 3m - 2)/2$
Abs/Dash/3	$nm + n - n$	$nm + n - (m^2 + 3m - 2)/2$
Abs/J/1	$2nm + 2n - n$	$nm + n - (m^2 - m - 2)/2$
Abs/J/2	$2nm + 2n - n$	$nm + n - (m^2 + 3m - 2)/2$
Abs/J/3	$2nm + 2n - n$	$nm + n - (m^2 + 3m - 2)/2$

3. WHAT IS POSSIBLE?

In this section we look at (i) a lower bound to the size of combinator code attainable for a *finite* set of combinators, and (ii) show that the problem of optimising combinator code using CL-Dash is NP-complete.

Theorem

Let CL-X be a combinatory logic, where the number of combinators in CL-X is finite. Then, for each abstraction algorithm Abs/X/r, there exist expressions E in CL-X with $|E| = n$ such that, if $x_1, \ldots, x_m$ are variables, then $|\text{Abs/X/r}(x_1, \ldots, x_m)(E)|$ is $\Omega(n*\log(m))$.

Proof

Assume CL-X contains p combinators. We enumerate the possible expressions of a given size n which contain only combinators ($\theta(p^n)$), and those which are allowed to contain up to m variables

$(\theta((m+p)^n))$. We argue that since Abs/X/r can take $\theta((m+p)^n)$ distinct function bodies as argument it must then return a similar number of distinct combinator expressions containing no variables. Thus these expressions containing no variables must be of a maximum length n′ where

$\theta(p^{n'}) = \theta((m+p)^n)$, thus $n' = \theta(n*\log(m))$. □

Next we show that optimising combinator code is NP-complete. The optimisation problem will be: given an expression E whose only atomic subexpressions are variables $x_1, \ldots, x_m$, and an integer q, does there exist an expression E′, whose only atomic subexpressions are combinators in CL-Dash, such that the expression $(E' x_1 \ldots x_m)$ reduces in q (or less) reduction steps to E? The NP-completeness of this problem will be proved by transformation from "Hitting Set" to a restricted set of such expressions.

The reduction strategy will be the equivalent of normal order for graphs, that is, leftmost-outermost first. The expression being reduced will be considered as a graph. Initially, at the start of the reduction, this graph will be a binary tree. Thereafter code-sharing will be allowed. OP, as defined above, can easily be shown to lie in NP, and we have the following result.

Theorem

The Optimisation Problem is NP-Complete.

Proof

Given in full in Ref. [7]. □

4. BALANCING

The concept of "balancing" an expression in order to improve the code obtained after abstraction of variables was introduced in Ref. [10].

Balancing is an operation which takes as input a combinatory expression, E, and returns as output an expression, F, in which may occur combinators, such that the resulting expression tree is partially "balanced"; then, when variables are abstracted from F, in the asymptotic case (|E| large, number of variables occurring in E large) the code produced will be shorter than if the variables were to be abstracted from E using the above abstraction algorithms.

It should be stressed that it is the asymptotic performance of an abstraction algorithm that is improved, and that for certain expressions the size of code produced will be worse.

It should also be stressed that balancing "works" for certain logics and abstraction algorithms only. For instance, CL-SKI together with Abs/SKI/1 will produce worse code after balancing has been performed in every instance. We examine only Abs/Dash/1 and Abs/Dash/2.

As Kennaway remarks in Ref. [11], the number of combinators produced when abstracting a variable from an expression E using Abs/Dash/1 is equal to the number of nodes in the minimal subtree of E containing all occurrences of that variable. When Abs/Dash/2 is used, a similar result holds, except that certain nodes are deleted.

Definition: Tsize

Let E be an expression in CL-Dash, considered as a binary tree, then tsize(E) is equal to the number of leaf nodes in E which are not combinators.

Definition: Select

Let E be an expression in CL-Dash. Then select(E) is a subexpression of E where $|\text{tsize}(E)/2 - \text{tsize}(\text{select}(E))|$ is minimised.

Definition: Balance

Let E be an expression in CL-Dash, F = select(E), v a variable not occurring in E, $E = (E_1\ E_2)$ and Abs/Dash/r the abstraction algorithm being used. Then balance(E) is defined as follows,

if $\text{tsize}(E) \leq 3$ then
balance(E) = E (i)

else

if $\text{tsize}(E)/3 \leq \text{tsize}(E_1) \leq 2*\text{tsize}(E)/3$ then

$\text{balance}(E) = (\text{balance}(E_1)\ \text{balance}(E_2))$ (ii)

else

$\text{balance}(E) = ((\text{Abs/Dash/r}(v)(\text{balance}([v/F]E)))\ (\text{balance}(F)))$ (iii)

For example, let $E = (x_1\ x_2\ x_3\ x_4)$, then

$$\text{balance}(E) = (\text{Abs/Dash/r}(v)(v\ x_3\ x_4)(x_1\ x_2)).$$

In Burton's original paper [10], the logic CL-Dash is used, together with the abstraction algorithm Abs/Dash/2.

Theorem

Let s be the maximum value of $|\text{Abs/Dash/1}(x_1, \ldots, x_m)(\text{balance}(E))|$, as E ranges over expressions in CL-Dash with $|E| = n$, and using Abs/Dash/1. Then

$$s \leq 6*n*\log(m) + 9*n - (\log(n) + \log(2/9))/(\log(3) - 1) - 10 \qquad (n \geq 2).$$

Once again, if balancing is performed using Abs/Dash/1 and using Abs/Dash/2, we find that the latter always produces more efficient code. However, their order of magnitude performance is similar—O(n*log(m))—and from section 3 we see that this is asymptotically optimal. So balancing using Abs/Dash/1 or Abs/Dash/2 yields code which is $\theta(n*\log(m))$.

Balancing can be considered as having two separate stages—that of balancing an expression, and that of abstracting variables from an expression after balancing it. The former behaves in an linear fashion, the latter in an "n*log(m)" way. The former is independent of which variables occur in the expression being balanced, the latter is by its very nature sensitive to how often the variables being abstracted occur in it.

It will be seen that the first process, that of balancing an expression, introduces combinators. For many practical purposes, this simply yields too many, giving code which is worse than that produced without balancing. In section 5 we give some numerical results on this.

Burton in Ref. [10] claims that, if the lengths of the variables used in the expression which is being balanced is taken into account in determining "size", then the growth in size of the expression is linear in the size of the original expression, thus, if $|E| = n$, using that definition of size, we get $|\text{Abs/Dash/2}(x_1, \ldots, x_m)(\text{balance}(E))|$ is O(n). We get a different result here, as we ignore the lengths of variables introduced.

5. IMPROVING CODE

In this section we shall examine heuristics for getting "better" code in CL-Dash than by simply using Abs/Dash/2. By this we shall mean: given an expression, E, containing variables $x_1, \ldots, x_m$, how can one produce an expression F such that (i) $x_1, \ldots, x_m$ do not occur in F and (ii) $(F\ x_1 \ldots x_m)$ reduces to E in a "small" number of reduction steps? The code improvements we display are not intended to be a definitive set, merely a collection which we have found to give significant improvements (at least, in certain cases).

We shall give examples of all these heuristics "in action", using sets of test functions. These will fall into two categories.

First we shall use a number of functions representing "real" problems. We do not know what "real" large functional programs will look like when such languages become more widely used, however the examples we use we consider typical.

Second, we present some functions which yield interesting results (but, again, may not occur in practice). These will be defined as lambda-expressions.

We shall assume that we have a function where any internal functions are considered as predefined atoms—thus we allow no global variables. We shall possibly alter that expression (for instance, by balancing it), then we shall translate it to an expression in CL-Dash.

Recursion will be implemented using the "least fixed-point" combinator Y, and we shall always be using acyclic graph structures. The definition of tsize treats as atomic a subexpression which

contains only constants. This seems reasonable on the grounds that such a subexpression represents a function which is effectively predefined, and into which combinators will not need to be inserted anyhow.

It often occurs that balancing an expression introduces more combinators than it gets rid of. We present first some methods of improving the balancing algorithm which do not compromise the asymptotic performance.

"B" shall refer to the balancing operation (using Abs/Dash/2), and will be used with a following digit.

"B0" will refer to balancing without any optimisations at all.

"B1" will alter case (iv) of the definition of balance to be:

if E contains more than one free variable then
  balance(E) = ((Abs/Dash/2(v)(balance([v/F]E)))(balance(F)))
else
  balance(E) = E.

"B2" will be defined similarly to "B1":

if F contains more than one free variable then
  balance(E) = ((Abs/Dash/2(v)(balance([v/F]E)))(balance(F)))
else
  balance(E) = ([F/v](balance([v/F]E))).

"B3" is similar to "B2", except that "full optimisation" is performed. That is, balance(F) and balance([v/F]E) are evaluated using "B3", the two cases above are both evaluated separately, the one yielding the shortest code after all variables have been abstracted then being chosen (in the ambiguous case balancing is not performed). It will be seen that "B3" must yield the shortest possible code consistent with the asymptotic analysis in section 4, since all possible combinations are tried. We do not suggest that "B3" be used in practice, but it sets a lower bound on the size of code obtainable by inhibitions of the balancing algorithm.

We note now that "B1" and "B2" must produce shorter code after variables have been abstracted using Abs/Dash/2 than "B0"; for, if select(E) contains at most one distinct variable, any extra variable inserted in place of select(E) must introduce more combinators (since any internal node which was previously "tagged" will remain so).

The "Bi" ($0 \leq i \leq 3$) are ordered, in that "Bj" produces shorter code than "Bi" if $j > i$. Since each of the optimisations produces better code than "B0" at each balancing step, the asymptotic performance of balancing is preserved and the worst case analysis of section 4 still holds.

The examples of programs given below are typical of a certain class of program, and are all fairly well balanced to start with—hence the poor results when the balancing algorithm is used. However, it can be argued that in certain cases badly balanced lambda-expressions might be created—for instance, in a compiler or similar program which includes large "case" statements. In such cases, inhibitions similar to "only balance if one immediate subexpression of E is at least ten times as big as the other"could be implemented. However, such a restriction would not be illustrated by any of our examples here. Indeed, even a naïve strategy such as "abstract without balancing, and also abstract with balancing, and choose whichever produces the shortest code" would not be infeasible. On the other hand, it could also be argued that the creation of badly-balanced lambda-expressions counts as "bad" programming style.

We perform essentially one sort of optimisation of the source lambda-expression apart from balancing. If we have an expression $E = (\gamma\, \alpha\, \beta)$ where γ is an arithmetic operator, α is a variable, and β is an integer, then we manipulate that expression as follows:

replace E by E′, where
  if $\gamma \in \{+, *, =\}$ then $E' = (\gamma\, \beta\, \alpha)$,
  else if $\gamma = -$ and β is an integer then $E' = (+ -\beta\, \alpha)$
  else if $\gamma = <$ then $E' = (>\beta\, \alpha)$
  else if $\gamma = \leq$ then $E' = (\geq \beta\, \alpha)$
  else if $\gamma = >$ then $E' = (<\beta\, \alpha)$
  else if $\gamma = \geqslant$ then $E' = (\leq \beta\, \alpha)$.

This optimisation is performed before any others. Note that we do not include corresponding replacements involving / and **.

We note that this optimisation will remove a number of C combinators from the abstracted code, since Abs/Dash/2(x)(γ x β) = (C γ β), and Abs/Dash/2(x)(γ β x) = (γ β). In the cases "A", "B2" and "B3", the optimisation will always improve the code. We denote these optimisations by prefixing "H" to the name of the following optimisation, thus "HB1" will mean: perform this high-level optimisation, then balance using "B1".

These "high-level" optimisations give significant code improvements, but are dependent on the symmetry of the arithmetic operators annotated to the combinatory logic. They therefore do not rely on properties inherent in the combinators being used. They may be considered as standing in close relation to established optimisation techniques for non-functional programs.

Now, let us consider the dashed combinators. The rules for Abs/Dash/2 say that the first combinator introduced when evaluating Abs/Dash/2(x)(γ α β), where γ is an expression which contains variables, but in which x does not occur, should be non-dashed, thus

Abs/Dash/2(x)(γ α β) = A(γ α)′ β′,

where

$\beta' \in \{\beta$,Abs/Dash/2(x_1)(β)$\}$, similarly (γ α)′, and A $\in$ {S, B, C}.

However, since x/oc γ, there may be circumstances in which it is desirable for the first combinator to be a dashed one, thus getting instead

(A′ γ α′ β′), where A′ $\in$ {S′, B′, C′}.

For example, consider abstracting a, b, c, d and e (in that order) from the expression

b(a b c e)(d e).

Using simply Abs/Dash/2, this becomes

(B′(B′ B) S(S(B′ B))),

however, using S′ when abstracting the variable e we get

(S(B′ S′)).

This is clearly a significant improvement. However, if we use this amendment to the definition of Abs/Dash/2 indiscriminately, then we may get a deterioration in performance. Abs/Dash/4, which we now define, is merely Abs/Dash/2 rewritten so that dashed combinators are always used when it is possible to use them. This is introduced not as an optimisation, rather to illustrate its performance.

Definition: Abs/Dash/4

As for Abs/Dash/2, except that k denotes an arbitrary expression in which x does not occur.

We now look at conditions for combining these two algorithms for abstracting variables (Abs/Dash/2 and Abs/Dash/4) so that improved code is produced. We shall amend the definition of Abs/Dash/4 so that a check is performed before using a dashed combinator to ensure that it really is a good idea. Essentially this check will consist of seeing whether introducing a dashed combinator will produce shorter code than not doing so by "counting" the combinators that will be introduced. The condition will be denoted by cond.

Definition: Abs/Dash/5

The first of the following which is applicable should be used (G is an expression in which x does not occur):

Abs/Dash/5(x)(x)	≡ I;
Abs/Dash/5(x)(E x)	≡ E if x/oc E;
Abs/Dash/5(x)(E)	≡ (K E) if x/oc E;

Abs/Dash/5(x)(G E F) ≡ (S G(Abs/Dash/5(x)(F)))
if E ≡ x and x oc F;

Abs/Dash/5(x)(G E F) ≡ (C G F)
if E ≡ x and x/oc F;

Abs/Dash/5(x)(G E F) ≡ (S′ G(Abs/Dash/5(x)(E))(Abs/Dash/5(x)(F)))
if x oc E, x oc F and cond(G E F);

Abs/Dash/5(x)(G E F) ≡ (C′ G(Abs/Dash/5(x)(E))F)
if x oc E, x/oc F and cond(G E F);

Abs/Dash/5(x)(G E F) ≡ (B′ G E(Abs/Dash/5(x)(F)))
if x/oc E, x oc F and cond(G E F);

Abs/Dash/5(x)(E F) ≡ (S(Abs/Dash/5(x)(E))(Abs/Dash/5(x)(F)))
if x oc E and x oc F;

Abs/Dash/5(x)(E F) ≡ (C(Abs/Dash/5(x)(E))F)
if x oc E and x/oc F;

Abs/Dash/5(x)(E F) ≡ (B E(Abs/Dash/5(x)(F)))
if x/oc E and x oc F.

Cond(G E F) is true if one of the following cases holds, otherwise false:

(i) G contains no variables;
(ii) G ≡ y for some variable y with y ≠ x;
(iii) G is not of the form (H L) where H contains no variables and E is not a variable;
(iv) G is not of the form (H L) where H contains no variables, E is a variable and E oc G;
(v) G ≡ (H y) for some variable y, some expression H containing no variables, and either E is not a variable or E ≡ y.

These conditions *ensure* that the code produced will be at least as short as that produced using non-dashed combinators. However, they make no assumptions about the order in which subsequent variables are abstracted. If we have access to this information, then we can improve on Abs/Dash/5 as follows:

Definition: Abs/Dash/6

As for Abs/Dash/5, except that clause (iv) and (v) of the definition of cond are replaced by

(vi) G is not of the form (H L) where H contains no variables, E is a variable, E oc G or E is abstracted after some variable which occurs in G has been;
(viii) G ≡ (H y) for some variable y, some expression H containing *no* variables, and either E is not a variable or E is not abstracted before y is.

We shall assume that our lambda-calculus contains constants. These will include integers, inequalities and " = " as integer relations, the empty list "nil", the list constructor ":", the list predicates "hd" and "tl", "null" which tests whether a list is empty or not, and the recursion combinator "Y". All functions will be Curried.

We use the following test functions:

(1) Ackerman's function (usual definition).
(2) Factorial (usual definition).
(3) The eighth Fibonnacci number—21 (usual definition).
(4) List of the first eight Fibonnacci numbers, using a cached list (21, 13, 8, 5, 3, 2, 1, 1).
(5) Knapsack—a version of the standard 0/1 knapsack problem. The first argument is an integer, the second is a list of integers.
(6) Permutations of a list—takes as input a list of integers, returns a list of permutations of that list.
(7) Powers23 is a function which returns a list of all numbers of the form $2^i + 3^j$ (i, j > 0).

(8) Primes—an infinite list, by test division. We take the fourth in that list. The list does not include 2.
(9) Primes—using the "sieve" technique.
(10) Bubblesort.
(11) Lopside-10.

$$((\lambda x_1 \ldots \lambda x_{10}.\ x_{10}(x_9(\ldots(x_2\ x_1)\ldots)))\ 1\ 2\ldots 9\ 10)$$

(12) Lopside-20.

$$((\lambda x_1 \ldots \lambda x_{20}.\ x_{20}(x_{19}(\ldots(x_2\ x_1)\ldots)))\ 1\ 2\ldots 19\ 20)$$

(13) Lopside-30.

$$((\lambda x_1 \ldots \lambda x_{30}.\ x_{30}(x_{29}(\ldots(x_2\ x_1)\ldots)))\ 1\ 2\ldots 29\ 30)$$

(14) Case-statement-10

$$((\lambda v.\ \lambda x_1 \ldots \lambda x_{10}.\ =v\ 1\ x_1(=v\ 2\ x_2(\ldots(=v\ 10\ x_{10}\ 11)\ldots)))\ 11\ 1\ldots 10)$$

(15) Case-statement-20

$$((\lambda v.\ \lambda x_1 \ldots \lambda x_{20}.\ =v\ 1\ x_1(=v\ 2\ x_2(\ldots(=v\ 20\ x_{20}\ 21)\ldots)))\ 21\ 1\ldots 20)$$

The results are given in Table 2, with the columns representing the number of the test function, as given in the previous section and the rows representing the number of combinator reduction steps.

The number of non-combinator (for instance, arithmetic) operations is not changed by these optimisations; we do not address the problem of amending an expression to alter the number of such operations.

Since we may wish to apply more than one optimisation to a particular lambda-expression, the names we shall give in these tables will contain the names of all the optimisations used. For instance, "B1HA2" will mean "balance using optimisation B1, then apply high-level optimisation H, then abstract variables using Abs/Dash/2".

Columns 1–10	"Real" examples
Columns 11–13	Contrived examples where balancing works well
Columns 14–15	Possible "real" examples where balancing works well

We do not include results for Abs/Dash/6, since they are identical in our examples to those for Abs/Dash/5.

Table 2.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A2	442	90	316	239	320	360	435	648	640	335	62	227	492	95	290
A4	400	90	316	239	306	360	427	635	626	325	62	227	492	140	480
A5	400	90	316	239	306	360	427	635	626	325	62	227	492	95	290
B0A2	669	119	612	369	599	553	636	1002	793	481	35	93	159	169	354
B1A2	669	111	571	349	599	533	636	956	784	481	35	93	159	169	354
B2A2	569	90	471	239	549	485	589	864	716	411	35	93	159	84	187
B3A2	442	90	316	239	327	360	435	648	640	335	35	93	159	84	187
HA2	362	75	235	212	320	360	435	633	618	335	62	227	492	85	270
HA5	320	75	235	212	306	360	427	620	604	335	62	227	492	85	270

6. CONCLUSIONS

The main conclusions to be drawn are as follows:

(i) Eta-abstraction does not significantly affect worst-case performance (although it affects beneficially the performance of abstraction algorithms for many "small' expressions).
(ii) The logics CL-Dash and CL-J produce worst-case code which is of size $\theta(n*m)$, where n is the size of the expression from which m variables are being abstracted, and although the other logics considered can be coerced into producing code with similar behaviour, the

method for doing it is at best long-winded and at worst produces code which is typically an order of magnitude worse than that produced from CL-Dash.

(iii) The "original" abstraction algorithm (Abs/Dash/2) gives remarkably good code, and attempts to improve it seem to have limited effect for the "real" functional programs that we have tried (improvement of around 10%).

(iv) We have an algorithm (balancing) which will produce code which is within a constant factor of the optimal, although its performance on "real" functional programs tends to be detrimental.

(v) The problem of optimising the code is NP-Complete, therefore we do not need to be disappointed that we have not found all the useful code optimisations.

Many unresolved questions remain. Results for other sets of combinators and for other abstraction algorithms have not been given. Our choice of combinators may well be far from optimal, and a different set of combinators might yield much improved code. In particular, we have not been concerned with combinatory logics which contain an infinite set of combinators. We do not, as yet, have many algorithms for improving combinator code, and more are needed.

7. SUMMARY

For several years now it has been recognised that combinators may be useful as a low-level code for functional programs. This idea has stimulated several papers, notably ones by Turner, and several machines (such as the Cambridge SKI machine). Some algorithms to translate a functional program (written as a lambda-expression) into combinators are examined, and "worst-case" analyses for these algorithms presented. These algorithms span several sets of combinators. When the set of combinators is finite, a lower bound to the performance of any abstraction algorithm is given. The problem of producing optimal code for the set of combinators introduced by Turner is shown to be NP-Complete. The balancing algorithm, originally devised by Burton, is examined, and shown to produce code which is within a constant factor of the optimal. The performance of balancing for "real" programs is looked at, and we show that balancing tends to have a detrimental effect. Methods of improving the performance of balancing are exhibited, together with some other algorithms for improving combinator code.

REFERENCES

1. Hindley J. R., Lercher B. and Seldin J. P., *Introduction to Combinatory Logic*. Cambridge University Press (1972).
2. Curry H. B., Craig W. and Feys R., *Combinatory Logic*, Vol. 1. North-Holland, Amsterdam (1968).
3. Seldin J. P. and Hindley J. R. (Eds), *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*. Academic Press, London (1980).
4. Turner D. A., A new implementation technique for applicative languages. *Software Pract. Exper.* **9**, 31–49 (1979).
5. Turner D. A., Another algorithm for bracket abstraction. *J. Symbolic Logic* **44**, 267–270 (1978).
6. Clarke T. J. W., Gladstone P. J. A., MacLean C. D. and Norman A. C., SKIM—the S,K,I reduction machine. *Conference Record of the 1980 LISP Conference*, Stanford University, California, pp. 128–135 (1980).
7. Joy M. S., On the efficient implementation of combinators as an object code for functional programs. Ph.D. Thesis, University of East Anglia (1985).
8. Kennaway J. R., Director strings as combinators. Internal Note, School of Computing Studies and Accountancy, University of East Anglia, Norwich (1982).
9. Garey M. R. and Johnson D. S., *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, San Francisco, California (1979).
10. Burton F. W., A linear space translation of functional programs to Turner combinators. *Information Processing Lett.* **14**, 201–204 (1982).
11. Kennaway J. R., The complexity of a translation of lambda-calculus to combinators. Internal Report CS/82/023/E, University of East Anglia, Norwich (1982).

About the Author—MICHAEL S. JOY was born in Manchester, England, in 1958, and received the degrees of B.A. and M.A. from the University of Cambridge and the degree of Ph.D. from the University of East Anglia. Dr Joy is currently a Lecturer at Wolverhampton Polytechnic. His research interests centre around declarative languages and their implementations.

About the Author—VIC RAYWARD-SMITH was educated at Tiffin School, Kingston and Hertford College, Oxford. Having obtained his B.A. in mathematics, he completed a postgraduate diploma in machine

intelligence at the University of Edinburgh and a Ph.D. from Queen Mary College, London. He is currently a senior lecturer at the University of East Anglia, Norwich. Dr Rayward-Smith has authored several undergraduate texts in computing and has research intersts in formal language theory, complexity theory and algorithm design and analysis.

About the Author—F. WARREN BURTON has a B.S. in applied mathematics and an M.A. in mathematics from the University of Colorado, and a Ph.D. in computing studies from the University of East Anglia. He has been on the faculty of Michigan Technological University and the University of East Anglia. He is currently an associate professor at the University of Colorado at Denver. Dr Burton is a member of ACM, Sigplan, the IEEE Computer Society and Sigma Xi. His research interests include functional programming, distributed computing, design and analysis of algorithms and computational geometry.