THE JOURNAL OF SYMBOLIC LOGIC
Volume 44, Number 2, June 1979

ANOTHER ALGORITHM FOR BRACKET ABSTRACTION

D. A. TURNER

This short article presents an algorithm for bracket abstraction [1] which avoids a combinatorial explosion in the size of the resulting expression when applied repeatedly for abstraction in a series of variables. It differs from a previous solution [2] in introducing only a finite number of additional combinators and in not requiring that all the variables to be abstracted be treated together in a single operation.

Introduction. We begin with a brief motivation for the benefit of the reader not familiar with the material in [1].

Much of the notation used in logic and mathematics can be recast in the following uniform syntax. A *term* is either an atom (i.e. a constant or a variable) or else it is of the form $A\ B$ where A and B are *terms* and their juxtaposition denotes the application of a monadic function (on the left) to its argument (on the right). We may use parentheses to resolve ambiguity together with the convention that, in the absence of parentheses, juxtaposition associates to the left. So in $A\ B\ C$ the function A is being applied to B and the result, itself presumably a function, is being applied to C. Functions of several arguments are adapted to this uniformly monadic syntax by replacing them by appropriately defined higher order functions of one argument. So for example a first-order dyadic function, d say, is replaced by a second-order monadic function d' defined so that $d'\ x\ y$ always has the same value as $d(x, y)$. (*Note*. This transformation is called "currying" after the logician H. B. Curry[1], thus we would say that d' is here a curried version of d.)

In addition to operations which can be construed as the application of a function to arguments, however, our customary notations also use constructions of an apparently very different character, namely those which introduce bound variables (e.g. the quantifiers, the integration sign, Church's λ). These too can be reduced to the above form provided that we can give a positive solution to the following problem. Given a *term*, X say, containing zero or more occurrences of the variable x, can we write down a *term* denoting the funtion f with the property (1) below?

(1) $$f x = X.$$

It turns out that provided we introduce some special constants (the *combinators*) we can always find an expression for this f. Specifically we can define an operation on *terms* (*bracket abstraction*) such that the result of using it on a *term* X with respect to a variable x, written $[x]\ X$, is a *term* containing no occurrences of x and denoting a function f with property (1) above. That is, it defines explicitly the

Received October 7, 1977.

[1] Strictly speaking inaccurately, since the earliest use of the idea is in [3].

function which we can write implicitly as $(\lambda x.X)$. Since all other variable-binding operations can be reduced to λ, see [1], they too can be eliminated. So by using bracket abstraction, together with currying, we can translate all our notation into a language in which the only operation is monadic functional application and in which bound variables do not occur. This very significant result was first discovered by Schoenfinkel [3].

Perhaps the simplest algorithm for bracket abstraction is as follows. Define the combinators $\boldsymbol{S}$, $\boldsymbol{K}$ and $\boldsymbol{I}$ by the properties (for arbitrary x, y, z)

$$\boldsymbol{S}\,x\,y\,z = (x\,z)\,(y\,z),$$

$$\boldsymbol{K}\,x\,y = x,$$

$$\boldsymbol{I}\,x = x$$

and define bracket abstraction as follows

$$[x]\,x = \boldsymbol{I},$$

$$[x]\,y = \boldsymbol{K}\,y,$$

$$[x]\,(AB) = \boldsymbol{S}([x]A)\,([x]B).$$

Here we use A and B to stand for arbitrary terms and y is a constant or a variable other than x. To prove the correctness of the algorithm, i.e. that the term $[x]\,X$ always denotes an f with property (1), appeal to the definitions of the combinators and carry out an induction on the size of the term X. (We leave this as an exercise for the reader.)

This basic algorithm tends to produce needlessly long-winded abstracts. Curry [1] gives an improved version which takes account of various special cases where the variable being abstracted does not occur in one or more subexpressions. Introduce the additional combinators $\boldsymbol{B}$ and $\boldsymbol{C}$ defined

$$\boldsymbol{B}\,x\,y\,z = x(y\,z), \qquad \boldsymbol{C}\,x\,y\,z = (x\,z)y$$

which are, so to speak, "degenerate" versions of $\boldsymbol{S}$. The improved algorithm for bracket abstraction can then be defined in terms of the basic one as follows. Use the basic algorithm but whenever a term of the form $\boldsymbol{S}\,A\,B$ is created apply one of the following optimisations if it is possible to do so

$$\boldsymbol{S}(\boldsymbol{K}\,A)\,(\boldsymbol{K}\,B) \to \boldsymbol{K}(A\,B),$$

$$\boldsymbol{S}(\boldsymbol{K}\,A)\,\boldsymbol{I} \to A,$$

$$\boldsymbol{S}(\boldsymbol{K}\,A)\,B \to \boldsymbol{B}\,A\,B,$$

$$\boldsymbol{S}\,A\,(\boldsymbol{K}\,B) \to \boldsymbol{C}\,A\,B.$$

Where more than one of the above rules is applicable the one given earlier takes precedence. The correctness of this new algorithm may be inferred from the correctness of the basic algorithm by observing that in each of the above rules the term on the right is *extensionally equal* to the original term on the left. (*Note*. We say that X is "extensionally equal" to Y when $X\,x = Y\,x$ for arbitrary x, i.e. just when their behaviour as *functions* is identical.)

By way of example the basic algorithm gives

$$[x]f\,x\,y = \boldsymbol{S}(\boldsymbol{S}(\boldsymbol{K}\,f)\boldsymbol{I})(\boldsymbol{K}\,y),$$

$$[y]f\,x\,y = \boldsymbol{S}(\boldsymbol{S}(\boldsymbol{K}\,f)\,(\boldsymbol{K}\,x))\,\boldsymbol{I}$$

whereas with the improved algorithm of Curry we have

$$[x]f\,x\,y = \boldsymbol{C}f\,y, \qquad [y]f\,x\,y = f\,x.$$

The new algorithm. In general the algorithm given by Curry produces compact results for the abstraction of a single variable from a term. When it is used repeatedly, to abstract on a series of variables in succession from a given term, however, the size of the resulting expression blows up as we can easily show. Suppose we are to abstract in turn the variables $x_1\ x_2 \cdots x_n$ from a term of the form $A\ B$. We use A' to mean $[x_1]A$, A'' to mean $[x_2]([x_1]A)$ and so on, similarly B', B'' and so on. Assuming for simplicity that each of x_1 to x_n occur at least once in both A and B, the Curry algorithm will yield the following results as we abstract on successive variables

$A\ B$	initial term,
$\boldsymbol{S}\,A'\,B'$	first abstraction,
$\boldsymbol{S}(\boldsymbol{B}\,\boldsymbol{S}\,A'')\,B''$	second abstraction,
$\boldsymbol{S}(\boldsymbol{B}\,\boldsymbol{S}(\boldsymbol{B}(\boldsymbol{B}\,\boldsymbol{S})A'''))\,B'''$	third abstraction,
$\boldsymbol{S}(\boldsymbol{B}\,\boldsymbol{S}(\boldsymbol{B}(\boldsymbol{B}\,\boldsymbol{S})\,(\boldsymbol{B}(\boldsymbol{B}(\boldsymbol{B}\,\boldsymbol{S}))\,A'''')))\,B''''$	fourth abstraction,

and so on, the size of the nth term rising at a rate which is at least quadratic in n, with disastrous consequences for any practical application in which we have to abstract on a large number of variables. It can be seen by inspection that what is happening is that each successive act of abstraction is introducing further complexity into the structure of the term which makes subsequent abstractions harder to perform. To prevent this from occurring we introduce the single new combinator $\boldsymbol{S}'$ defined by

$$\boldsymbol{S}'\,k\,x\,y\,z = k\,(x\,z)\,(y\,z)$$

which is just like $\boldsymbol{S}$ except that it "reaches across" an extra term at the front. By using $\boldsymbol{S}'$ instead of $\boldsymbol{S}$ for the second and subsequent abstractions we can leave the structure of the abstract unaltered in the form $\kappa\ \alpha\ \beta$ where κ is a term composed entirely of combinators, thus

$A\ B$	initial term,
$\boldsymbol{S}\,A'\,B'$	first abstraction,
$\boldsymbol{S}'\,\boldsymbol{S}\,A''\,B''$	second abstraction,
$\boldsymbol{S}'(\boldsymbol{S}'\,\boldsymbol{S})\,A'''\,B'''$	third abstraction,
$\boldsymbol{S}'(\boldsymbol{S}'(\boldsymbol{S}'\,\boldsymbol{S}))\,A''''\,B''''$	fourth abstraction,

and so on, the sizes of the successive terms now forming only a linear progression (the constant term at the front simply increases by one $\boldsymbol{S}'$ at each abstraction).

As a further refinement, to deal efficiently with the case that a variable does not occur in one or other of the subexpressions we can introduce the combinators $\boldsymbol{B}'$ and $\boldsymbol{C}'$ which are to $\boldsymbol{S}'$ exactly as $\boldsymbol{B}$ and $\boldsymbol{C}$ are to $\boldsymbol{S}$. Thus

$$\boldsymbol{B}'\,k\,x\,y\,z = k\,x(y\,z), \qquad \boldsymbol{C}'\,k\,x\,y\,z = k(x\,z)\,y.$$

The new definition of bracket abstraction can then be given as follows. Use the algorithm of Curry but whenever a term beginning in $\boldsymbol{S}$, $\boldsymbol{B}$ or $\boldsymbol{C}$ is formed use one of the following transformations if it is possible to do so

$$\boldsymbol{S}(\boldsymbol{B}\,K\,A)\,B \to \boldsymbol{S}'\,K\,A\,B,$$
$$\boldsymbol{B}(K\,A)\,B \to \boldsymbol{B}'\,K\,A\,B,$$
$$\boldsymbol{C}(\boldsymbol{B}\,K\,A)\,B \to \boldsymbol{C}'\,K\,A\,B.$$

Here A and B stand for arbitrary terms as usual and K is any term composed entirely of constants. The correctness of the new algorithm can be inferred from the correctness of the Curry algorithm by demonstrating that in each of the above transformations the left- and right-hand sides are extensionally equal. In each case this follows directly from the definitions of the combinators involved.

The application of bracket abstraction with which the author has been concerned is in taking a very high level computer programming language [4] in which one defines recursive functions by writing down equations involving bound variables, and compiling it to a form in which it can be executed by an extremely simple computer whose instruction set consists of the combinators. The above modifications to the abstraction algorithm were found to be necessary to keep the size of the compiled code within acceptable bounds.

Dr. P. H. Welch of the University of Kent has drawn the author's attention to the work described in [2] which defines a multiple abstraction operator which abstracts on any number of variables simultaneously. While this can produce more compact code than repeated application of the single abstraction operation given here, it proved less suitable for the author's purpose. First, because the algorithm for multiple abstraction uses an infinite number of combinators (via the schemata $\boldsymbol{K}_n\ \boldsymbol{I}_n^m\ \boldsymbol{B}_n^m$, see [2]) which would give the proposed target computer an undesirably complicated instruction set. Second, because in the compilation process the need to abstract on different variables arises at successive stages and rather than an algorithm for multiple abstraction as such it is more convenient to have a definition of simple abstraction which is well behaved under self-composition, as here.

REFERENCES

[1] H. B. Curry and R. Feys, ***Combinatory logic***, vol. 1, North-Holland, Amsterdam, 1958.

[2] S. K. Abdali, *An abstraction algorithm for combinatory logic*, this Journal, vol. 41(1976), pp. 222–224.

[3] M. Schoenfinkel, *Über die Bausteine der mathematischen Logik*, ***Mathematische Annalen***, vol. 92(1924), pp. 305–316.

[4] D. A. Turner, SASL language manual, St. Andrews University, Fife, Scotland, 1976.

COMPUTER LABORATORY
UNIVERSITY OF KENT
CANTERBURY, ENGLAND