SUPER-COMBINATORS

A New Implementation Method for Applicative Languages

R. J. M. Hughes

Oxford University Programming Research Group

United Kingdom

Introduction

There is a growing interest nowadays in functional programming languages and systems, and in special hardware for executing them. Many of these implementations are based on a system called graph reduction (GR), in which a program is represented as a graph which is transformed, or reduced, by the machine until it represents the desired answer. The various graph reduction implementations differ in the structure of the "machine code" (the program graph) and the compilation algorithms necessary to produce it from a source language. This paper describes a new implementation method using super-combinators which is apparently more efficient than its predecessors. Consideration of the new method also helps clarify the relationships between several other graph-reduction schemes. This paper is necessarily brief, but a fuller account can be found in (Hughes).

The simplest machine language we shall consider consists of constants combined by function application. This is the language of constant applicative forms (cafs). Some of the constants are basic functions, curried if necessary, such as +. The machine operates by applying basic functions to their arguments. An attractive feature of this language is that after an application has been evaluated it can be overwritten by its new value. This ensures that every expression is evaluated at most once. This important property is called *full laziness*. It is related to *lazy evaluation* of the λ -calculus [Henderson & Morris], with which every expression passed as a parameter to a λ -expression is evaluated at most once. So fully lazy evaluation implies lazy evaluation, but not vice versa.

It is obvious that the language of cafs is adequate for compiling arithmetic expressions into, for example, but it is not obviously sufficiently powerful to express an arbitrary functional program. However, any functional program can easily be translated into the λ -calculus. Henceforth we take the λ -calculus as the canonical functional programming language.

The λ -calculus can be used as a graph reduction machine language by introducing variables and λ -binding into the language of cafs. λ -expressions are treated as non-basic functions, and application of a λ -expression means copying the body, substituting the actual parameter for occurences of the bound variable. Because expressions are

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

© 1982 ACM 0-89791-082-6/82/008/0001 \$00.75

copied in this approach, they can be evaluated more than once. So evaluation may not be fully lazy. Substitution into expressions is inefficient in any case, because it must visit every leaf. An alternative is to avoid copying expressions by associating an environment, consisting of delayed substitutions, with each expression. But now the same expression can take different values in different environments, and so it is no longer possible to overwrite expressions with their new values. Because of this expressions may still be evaluated more than once and evaluation may not be fully lazy. In addition the manipulation of environments can be very expensive in its own right. So it appears that λ -expressions do not make good operators for a GR machine.

Fortunately, the λ -calculus can be translated into cafs if three new basic operators, S, K, and I are added [Curry & Feys]. These new operators are known as combinators. A translation into cafs using these combinators leads to an exponential growth in program size, but Turner has shown that if a few more combinators are introduced in the same style, and if a number of optimisation rules are applied during translation, then the growth in code size becomes quite manageable [Turner]. Moreover, Turner found that his optimisation rules led to a fully lazy evaluation of the original program when used in a GR implementation such as we have described. He found such an implementation to be considerably more efficient than a direct λ -reduction implementation.

However, with this approach the machine code form of a program is far removed from the source form. This makes interpretation of intermediate values difficult during debugging. Compilation is slow, both because of the number of optimisation rules that have to be applied, and because every λ -expression is compiled independently. This leads to many passes over expressions nested within many λ -expressions. Thirdly, execution is broken down into very small steps, making the overhead of linking one step to the next considerable. The remainder of this paper describes an approach that overcomes these disadvantages to some extent.

Introducing Super-combinators

The key to the new approach is to generalise the class of combinators. Recall that S, K, and I can be defined by λ -expressions:

$$S = \lambda x \lambda y \lambda z. \quad (x z) \quad (y z)$$

$$K = \lambda x \lambda y. \quad x$$

$$I = \lambda x. \quad x$$

These λ -expressions have two special properties that make them suitable for use directly as operators in a graph reduction machine. Firstly, they have no free variables and so are "pure code", hence their internal structure is of no consequence and any suitable representation may be used for them. Secondly, their bodies are applicative forms, ie are composed from variables and constants by application. This means that when cals are substituted for their bound variables, the result is a cal. If they are to be used as operators in a cal reduction machine this property is vital. Any λ -expression with these two properties is a combinator, and henceforth it is assumed that any combinator is a suitable operator for a cal reduction machine. Where it is necessary to distinguish generalised combinators from Turner's, they are called super-combinators. (Super-combinators are closely related to "proper combinators" [Curry & Feys], but differ from these in that constants may occur in their bodies, and in that combinators themselves are regarded as constants).

As there are infinitely many possible combinators, the GR machine will not contain definitions of them all at once. A compiler must generate definitions for the combinators it uses in the program graph. These definitions will be presented in this paper as λ -expressions or as equations, but in practice would probably be compiled into something

else, for example microcode. A compiler for such a machine must translate general λ -expressions into applicative forms. This can be done very simply as follows. Take any λ -expression, λV . E. First the body is converted into an applicative form by invoking the compiler recursively, giving λV . E'. Then the free variables of the λ -expression are identified. Suppose they are P, Q, ..., R. The λ -expression is prefixed by a λ binding each free variable, giving

$$\lambda P \lambda Q \dots \lambda R \lambda V. E'$$

The λ -expression resulting is a combinator, because E' is an applicative form and all its free variables P, Q, R are bound. Call this combinator α . Its defining equation is

$$\alpha P Q \dots R V \rightarrow E'$$

Then the original λ -expression is equivalent to, and can be replaced by $(\alpha \ P \ Q \ \dots \ R)$. When this form is applied to an expression E, the application of α can be reduced. E is bound to V, and each free variable takes its own value in E', so the applicative form is truly equivalent to the original λ -expression.

As an illustration, consider the source language definition

el n s = if n = 1 then hd s else el
$$(n - 1)$$
 $(tl s)$ fi

This defines the function el, which selects the nth element from the sequence s. In the λ -notation it is

el =
$$Y$$
 (λ el λ n λ s. /F (= n l) (hd s) (el (- n l) (tl s)))

Consider first the innermost λ -expression.

$$\lambda s. \ IF \ (= n \ 1) \ (hd \ s) \ (el \ (- n \ 1) \ (tl \ s))$$

Its free variables are n and el, so the combinator α is introduced with the definition

$$\alpha$$
 n el s \rightarrow /F (= n 1) (hd s) (el (- n 1) (tl s))

Now the whole λ -expression can be replaced by $(\alpha \ n \ el)$ and so

el =
$$Y$$
 (λ el λ n. α n el)

Repeating the process, combinators β and γ are introduced defined by

$$\beta$$
 el n \rightarrow α n el

$$\gamma$$
 el $\rightarrow \beta$ el

and so finally el is equal to $(Y \gamma)$.

It is clear by inspection that this choice of combinators is not optimal. The most serious problem, though, is not obvious. In fact, this compilation algorithm does not give a fully lazy implementation. To see this, consider the partial application (el 2), ie the function that returns the second element of a sequence.

el 2 =
$$\lambda s$$
. IF (= 1 2) (hd s) (el (- 2 l) (tl s))

This is equivalent to

el 2 =
$$\lambda s$$
. IF-FALSE (hd s) (el l (tl s))

where IF-FALSE is defined by

$$\mathit{IF-FALSE}\ E_{\mathsf{true}}\ E_{\mathsf{false}}\ o\ E_{\mathsf{false}}$$

Of course, in a fully lazy implementation, this is what (e1 2) would become after one application. The expressions $(IF (= 1 \ 2))$ and (e1 $(- 2 \ 1)$) would be evaluated only once. However, applying the combinators derived above we find

el 2
$$\rightarrow$$
 Y γ 2

 \rightarrow γ el 2

 \rightarrow β el 2

 \rightarrow α 2 el

and no further reduction is possible until s is supplied. A separate copy of the expressions mentioned above is created each time α is applied, and so they must be evaluated more than once. This is not a fully lazy implementation.

A Fully Lazy Super-combinator implementation

Fortunately the expressions subject to such repeated evaluation are easily identified. Any sub-expression of a λ -expression which does not depend on the bound variable risks it. Such expressions are called the *free expressions* of the λ -expression by analogy with free variables. Free expressions which are not part of any larger free expression are called *maximal free expressions* of the λ -expression.

The translation scheme given above converts the *minimal* free expressions of each λ -expression into parameters of the corresponding combinator. Consider a scheme which converts the *maximal* free expressions into parameters instead. First we must establish that this is a valid translation scheme, *ie* that genuine combinators are produced and that each λ -expression is replaced by an applicative form. Let us consider the application of the new scheme to a single λ -expression whose body is already an applicative form. The combinator produced must satisfy the definition, *ie* its body must be an applicative form and it must have no free variables. Its body will certainly be an applicative form, because it is derived from an applicative form (the body of the original λ -expression) by substituting new parameter names for certain expressions. It can have no free variables because any free variable must be part of some maximal free expression, and so will be removed as part of a parameter. So a genuine combinator is produced. The final result which replaces the original λ -expression is just the new combinator applied to the maximal free expressions, each of which was already an applicative form. So the λ -expression is replaced by an applicative form. Therefore this new translation scheme is valid.

Applying it to the el example,

$$\lambda s. \ /F \ (= n \ 1) \ (hd \ s) \ (el \ (- n \ 1) \ (tl \ s))$$

has maximal free expressions (IF (= n 1)) and (e1 (- n 1)). So a new combinator α is defined by

$$\alpha p q s \rightarrow p (hd s) (q (tl s))$$

and the definition of el becomes

el = Y (
$$\lambda$$
el λ n. α (IF (= n l)) (el (- n l)))

Continuing the process, β and γ are defined by

$$\beta$$
 el n \rightarrow α (/F (= n 1)) (el (- n 1))
 γ el \rightarrow β el

and el is equal to $(Y \gamma)$ as before.

Now, reconsider the partial application (el 2). With the new combinators

el 2
$$\rightarrow$$
 Y Y 2
 \rightarrow Y el 2
 \rightarrow β el 2
 \rightarrow α (/F (= 2 1)) (el (- 2 1))

Now whenever (el 2) is used, the same copies of the free expressions are used and hence they are evaluated only once. In fact,

el 2
$$\rightarrow$$
 α IF-FALSE (α IF-TRUE (el (- 1 1)))

and it will be reduced to this on the first call. In this example the new scheme gave fully lazy evaluation. In fact it does so in general.

A slight modification is advisable in a real compiler to improve the treatment of constant expressions. According to the description above, constant expressions are free expressions and so will be exported as parameters. It is an unnecessary overhead to pass constants from one place to another in this way. If the application of a combinator is redefined so that constant sub-expressions of the body are not recreated on each call, but a pointer to a single version used instead, then it is unnecessary to export such sub-expressions. From now on it is assumed that combinator application in defined in such a manner.

This compilation scheme can be implemented by a recursive tree walk. The body of each λ -expression is scanned, and each expression classified as constant, free, or variable (ie dependent on the bound variable). The maximal tree expressions are easily identified during the tree walk, and, unless they are constant, the compiler exports them as follows. The newly found mfe (maximal free expression) is compared with each previously found one. If it is the same as any of them then it is replaced by the corresponding parameter name. Otherwise a new parameter name is allocated and the mfe replaced by it. The compiler records the correspondence between parameter names and mfes. When the whole body has been scanned the compiler can generate code for the new combinator it has found, and use the combinator to construct the replacement applicative form as described above.

This scheme is the most advantageous one so far described. Evaluation is fully lazy, as in Turner's method. Compilation is faster, partly because the algorithm is simpler, and partly because the expressions replacing λ -expressions are in general considerably smaller then the original λ -expression, whereas using Turner's combinators they are about the same size. This is because the part of the expression that becomes the new combinator definition is thereby removed from further consideration. Parts of the program may still be scanned many times, but because they shrink on each scan this is not nearly so serious as in Turner's scheme. The code generated is close to the program source because each combinator corresponds directly to a source λ -expression. This makes the interpretation of intermediate states easier. Finally, execution steps are large and so the overheads of linking each step to the next are less significant. Experiments show that this method does indeed give faster execution than Turner's, which in turn is faster than the other methods.

Ordering Super-combinator Parameters

Even this scheme generates sub-optimal combinators though. Recall that γ was defined above by

$$y \ el \rightarrow \beta \ el$$

Clearly, γ and β have exactly the same effect and there is no need for a separate combinator γ at all. It is reasonable to expect a compiler to detect such redundant combinators and eliminate them. Fewer combinators means fewer reductions to be performed, and hence more speed.

Even had y been defined by, say,

$$\gamma$$
 n el $\rightarrow \beta$ (x n 2) el

it would be reasonable to expect a compiler to simplify γ by dropping the redundant parameter e1, so that γ would be defined by

$$\gamma n \rightarrow \beta (\times n 2)$$

The effect of this change is to allow γ to be applied earlier, when fewer parameters are available, and hence for its result to be shared more widely. Sharing the result more widely means that γ itself will be called less often, and hence saves time. It will be assumed that the compiler detects such redundant parameters and combinators and optimises them out, but without taking any further interest in how it does it.

Recall two more definitions that appeared above,

$$\beta$$
 el n \rightarrow α n el α n el s \rightarrow ...

In this case no parameters appear to be redundant. However, n and e1 were introduced as parameters of α by the compiler itself, because they were maximal free expressions. The *order* in which these parameters occur has so far been left completely arbitrary. Had the compiler arranged them in the other order the definitions would appear as

$$\beta$$
 el n \rightarrow α el n α el n α el n s \rightarrow ...

and now β is the same combinator as α and can be eliminated. It follows that some orders of parameters permit more optimisation than others and the compiler should choose an order allowing maximum optimisation.

The order chosen for combinator parameters also affects the compilation of enclosing λ -expressions. The applicative form replacing each λ -expression may well contain free expressions of the next enclosing λ -expression. The order of the combinator parameters will affect the size and number of these free expressions, and should be chosen to make them as large and few as possible. The larger free expressions are, the earlier large expressions are created and so the more widely they are shared. The fewer free expressions are, the fewer parameters their enclosing combinators have, and the more efficient those combinators are.

For example, consider the applicative form $(\alpha \ (hd \ s) \ n \ (tl \ s))$ in which the parameters of α can be arranged in any order. If the immediately enclosing λ -expression binds n then the maximal free expressions of the form as it stands are $(\alpha \ (hd \ s))$ and $(tl \ s)$. However, if it were rearranged as $(\alpha \ (hd \ s) \ (tl \ s) \ n)$ then the only maximal free expression would be $(\alpha \ (hd \ s) \ (tl \ s))$.

If. on the other hand, the immediately enclosing λ -expression bound s then the optimal ordering of the parameters would be $(\alpha \ n \ (hd \ s) \ (tl \ s))$ making $(\alpha \ n)$ the only maximal free expression. So, to maximise the size and minimise the number of *mies* of the next enclosing λ -expression, all the *mies* of the λ -expression being compiled which are also free expressions of the next enclosing λ -expression must appear before those which are not.

Suppose a λ -expression is being replaced by (α E_1 ... E_n). Then there should be some j such that for all i less than or equal to j, E_i is a free expression of the next enclosing λ -expression, and no E_i with i greater than j is. This guarantees that (α E_1 ... E_j) will also be a free expression of the next enclosing λ -expression. Now consider the λ -expression enclosing that. To maximise the size of its mies in the same manner all the E_i which are mies of it should appear before the E_i which are not, and so on and so forth. In general, the optimal ordering of the parameters under this criterion can be established as follows. Every E_i is a free expression of the λ -expression being compiled, but it may also be a free expression of one or more enclosing λ -expressions. Call the innermost λ -expression of which E_i is not a free expression its native λ -expression. If the native λ -expression of E_i encloses the native λ -expression of E_i then E_i precedes E_i in the optimal ordering. This does not necessarily define the optimal ordering uniquely, because expressions with the same native λ -expression may occur in any order. However, any ordering satisfying this condition is as optimal as any other.

Notice that an expression has no meaning outside its native λ -expression, because, by definition, the bound variable of its native λ -expression occurs in it somewhere. Notice also that constant expressions have no native λ -expression, because they are free in all λ -expressions. For the sake of uniformity they are assumed to be native

to some notional λ -expression enclosing the whole program and binding the names of all constants.

Now, having deduced an optimal ordering from our second criterion, let us return to our first: the compiler should arrange the parameters so as to allow maximum elimination of redundant parameters. The compiler only has any choice in the matter in the case of one combinator defined directly as a call of another. For example, consider

$$\beta$$
 pqrs $\rightarrow \alpha \dots s \dots$

No parameter is redundant unless the last one is, but the last parameter of a combinator must always be the bound variable of the λ -expression it was derived from. This means that, in the example, s was the bound variable of the λ -expression immediately enclosing α . If the parameters of α have been ordered optimally as defined above, then all parameters involving s come at the end of its parameter list. If there is only one such parameter, and it is s itself, then s is a redundant parameter of β and can be eliminated. Now, the call of α must have taken the form $(\alpha \ E_1 \ \dots \ E_n \ s)$ where s did not occur in any of the E_i . This means that each E_i is free in β , and hence so is all of $(\alpha \ E_1 \ \dots \ E_n)$. So in fact, if there are any E_i then β must have been defined by

$$\beta$$
 ps \rightarrow ps

where p corresponds to $(\alpha \ \mathbf{E_1} \ ... \ \mathbf{E_n})$. If α had only s as a parameter then β must have been defined by

$$\beta s \rightarrow \alpha s$$

In the first case β is equal to I. The λ -expression β is being generated from will be replaced by the application (β (α E_1 ... E_n)) ie by (I (α E_1 ... E_n)). β might as well be eliminated entirely and the λ -expression replaced by (α E_1 ... E_n) directly. In the second case β is equal to α . So we see that the optimal ordering derived from our second criterion also satisfies our first, and moreover it makes the job of detecting redundant parameters particularly simple.

Let us return to the example of el and compile it once more. el is defined by

el = Y (
$$\lambda$$
el λ n λ s. IF (= n l) (hd s) (el (- n l) (tl s)))

The innermost λ -expression has two *mfes* (*IF* (= n 1)) and (e1 (- n 1)). Make these into parameters p and q. Both these *mfes* have the same native λ -expression, so their order is immaterial. α can now be defined by

$$\alpha p q s \rightarrow p (hd s) (q (tl s))$$

and so

el =
$$Y (\lambda e \lambda n. \alpha (IF (= n 1)) (el (- n 1)))$$

Now, the next λ -expression has only one mfe, el. So β can be defined by

$$\beta$$
 el n \rightarrow α (F (= n 1)) (el (- n 1))

making

el =
$$Y (\lambda el. \beta el)$$

y would be defined by

$$\gamma$$
 el \rightarrow β el

and so as γ is equal to β it will not be generated. Finally e1 is equal to $(Y \beta)$.

A Compilation Algorithm which Orders Parameters Optimally

The next step is to design a compilation algorithm to order combinator parameters optimally. The algorithm will need to know the native λ -expression of each maximal free expression to select this order. But note that the λ -expressions enclosing any point can be identified by the number of λ -expressions enclosing them. Thus the outermost λ -expression is identified by zero, the next outermost by one, etc. Let us represent the λ -expressions enclosing each point by

these numbers.

The compiler must compute the identifying number of the native λ -expression of each expression in the program. In the case of variables, this is easy: the native λ -expression of a variable is the λ -expression binding it. Constant expressions have no native λ -expression, but can be assigned the number -1 to signify an all-encompassing λ -expression which encloses the whole program and binds all constants. Now, consider the application of a function to an argument, each of whose native λ -expression number is known. Because the function and argument appear at the same point, one native λ -expression encloses the other. Both function and argument are free expressions of any λ -expression enclosed by both native ones, and so the whole application is too. One or the other of the function and argument is *not* free in the innermost native λ -expression, though, so nor is the application. This means that the native λ -expression of an application is the innermost of the native λ -expressions of the function and argument. In terms of the identifying numbers, the number of an application is the *maximum* of the numbers of the function and argument. Taking advantage of these facts a compiler can easily compute these numbers for every expression in the program in a single pass.

Furthermore, those expressions which are maximal free expressions of any λ -expression can be identified at the same time. They are the expressions whose native λ -expression encloses the native λ -expression of the next larger expression. Such expressions are maximal free expressions of the native λ -expression of the next larger expression. They are certainly free in it, because any expression is free in all λ -expressions enclosed in its native one by definition. But the next larger expression is not free in it, because an expression is not free in its native λ -expression. Hence the first expression is maximal free. In terms of numbers, the maximal free expressions can be identified as those whose numbers differ from the number of the next larger expression.

So, during one pass over the program the compiler can identify all maximal free expressions and decide which λ -expression each one is an *mfe* of. During the same pass it might as well replace *mfes* by parameter names. Now, after the body of a λ -expression has been completely scanned, all *mfes* of that λ -expression have been identified and replaced by parameter names. The compiler can now generate the optimal combinator provided it can arrange the parameters in the right order. To do this it must decide which native λ -expressions of *mfes* enclose which others. But the nesting depth of each native λ -expression is being used to identify it and has already been computed, so to decide whether one native λ -expression encloses another the compiler need only compare these numbers. Using this the optimal combinator can easily be generated. Finally the applicative form to replace the λ -expression can be constructed and the native λ -expression of each part of it computed at the same time. The compiler can then continue to scan the rest of the program.

This algorithm was suggested by the concept of native λ -expression and the observation that the λ -expressions enclosing any expression can be identified by nesting depth. It generates optimal combinators according to the criteria we developed. Not only that, it accomplishes this in a *single* pass over the program, something no other compilation algorithm above was able to do. This makes it the fastest algorithm in this paper, as the compilation time is roughly linear in the size of the program.

Experimental Results

The proof of the pudding, though, is in the eating. To test these ideas in practice a compiler was written which compiled a high-level functional language into the λ -notation, and then could translate the result either into Turner's combinators or into super-combinators as required. Turner's combinators were selected for the comparison because

he had already found them to be superior to direct λ -reduction. A graph reduction program was written which contained definitions of all Turner's combinators, and was able to load and use definitions of super-combinators written in BCPL. The compiler produced its definitions of super-combinators in BCPL so that they could be compiled by the BCPL compiler and used by the reducer. Ten test programs were written, ranging in length from a few lines to a page and in purpose from the computation of e to twenty places to unification. They were each compiled to both kinds of code and run by the reducer. Measurements were made during compilation and execution and the results were as follows. The expected improvement in compilation time did not manifest itself for the smaller programs, some of which were compiled 50% more slowly into super-combinators. The larger programs were compiled more quickly, with the largest gaining 40%. Doubtless this advantage would be still more for even larger programs. The storage requirements at run-time were measured both as the total number of cells allocated during execution, and as the maximum number of cells required at one time. They seemed to be approximately the same in both implementations. except in the case of the program for computing e which consumed almost twice as many cells when run using Turner's combinators as when using super-combinators. The super-combinator code showed a consistent speed advantage, ranging from unmeasurably small for some of the smaller programs to 45% for the largest. So it seems that super-combinators have a moderate, but not phenomenal, advantage over Turner's combinators on efficiency grounds.

Graphical Combinators and Other Improvements

Implementation has not progressed beyond this point, but two avenues for improvement are being contemplated. The first and simplest is to order the parameters of commutative operators in the same way as the parameters of combinators. The same benefits should accrue. In fact, it might be worthwhile to have several versions of each non-commutative operator so that parameters of these can be ordered optimally too.

The second is suggested by the observation that application of a combinator involves constructing a tree incorporating the parameter values. There is no reason why a more general graph should not be constructed instead. The difference between a graph and a tree is that some parts may be shared between several branches, and some pointers may be circular. Such graphs may be described on paper by notation of the form

let
$$I = E_1$$
 in E_2

meaning the graph obtained from E_2 by substituting a pointer to E_1 for all occurrences of I in E_1 and E_2 . As an example of a combinator with a graph as its body, recall that $(Y \ f)$ reduces to the graph

Therefore

$$Y f = let x = f x in x$$

Of course, this syntax resembles the declarations of functional programming languages very strongly. Indeed, any style of programming language declaration can be translated easily into this form. So, declarations, which were previously interpreted by a translation into the λ -notation, can be reinterpreted as graph-structured programs. Such programs will contain fewer, but larger, λ -expressions to be converted into combinators. Provided that the algorithms in this paper are extended to deal with graphs instead of trees, combinators with graphs as bodies can now be generated. Fewer and larger combinators will be produced. This will reduce the overheads of linking one combinator to the next still further. The extension of the compilation algorithms to graphical programs should present no major problems.

Conclusion

To summarise, the λ -notation was taken as the canonical functional programming language, and the language of constant applicative forms was chosen as a graph-reduction machine-code. A translation of the λ -notation to cats was exhibited (Turner's combinators) and it was shown that the caf code was fully lazy while the original λ -notation was not. Turner's combinators achieved full laziness by breaking the computation down into very small, independent steps, which was not conducive to efficiency or clarity. A scheme was desired which would achieve full laziness more directly, in a clear and efficient way. Such a scheme was found and examined in some depth. The question arises of whether a λ -reducer could be modified to achieve full laziness more directly still.

It has been shown that full laziness is achieved provided the maximal free expressions of a λ -expression are not copied when the λ -expression is applied. This was arranged in the super-combinator approach by exporting them as parameters, so that they were no longer a part of the combinator body and so could not be copied. If, in a λ -reducer, the maximal free expressions of each λ -expression were marked in some way, and the substitution of a value for the bound variable did not copy them, then the λ -reduction would be done in a fully lazy way. This provides yet another way of achieving fully lazy evaluation.

Now the relationship between the different schemes is clear. Ordinary λ -reduction is not fully lazy, but can be made so in a fairly simple way. This is really an interpretive implementation, because *mfe* markers must be present at run-time and interpreted during substitution. Super-combinators provide a compiled implementation of the same scheme, because *mfes* have been recognised at compile-time and have no significance at run-time. Super-combinator code can be run on a *caf* reduction machine. Turner's combinators execute the program in the same way as super-combinators, but perform each super-combinator as a sequence of simple combinators. Turner's combinators can be run on a *caf* reduction machine with a fixed number of operators.

Acknowledgements

I wish to thank Chris Dollin, Peter Henderson, Geraint Jones, and Bernard Sufrin for their helpful comments on this work, and the Science and Engineering Research Council of Great Britain for their support.

References

[Curry & Feys]H.B.Curry & R.Feys: "Combinatory Logic", North-Holland
Publishing Company, Amsterdam, 1958.

[Henderson & Morris]P.Henderson & J.H.Morris: "A Lazy Evaluator", Proc 3rd annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, Atlanta, 1976 pp95-103.

(Hughes) R.J.M.Hughes: "Graph-Reduction with Super-combinators".
Oxford University Programming Research Group technical monograph PRG-28 (to appear).

[Turner] D.A.Turner: "A New Implementation Technique for Applicative Languages", Software, Practice and Experience, Vol. 9, 1979.