GRAPH REDUCTION WITH SUPER-COMBINATORS

BY

JOHN HUGHES

Technical Monograph PRG-28
June 1982

Oxford University Computing Laboratory
Programming Research Group
45 Banbury Road
Oxford OX2 6PE

Oxford University Computing Laboratory
Programming Research Group
45 Banbury Road
Oxford OX2 6PE

Abstract

This paper explains the principles of graph reduction and develops a new graph reduction method based on super-combinators. An optimality criterion for the new method is derived and a simple method of generating optimal code presented. The results of an experimental comparison are reported, showing the super-combinator method to be more efficient than previous ones.

Contents

Page

Acknowledgements

I wish to thank Chris Dollin, Peter Henderson, Tony Hoare, 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.

Introduction

There is a growing interest nowadays in functional programming languages and systems, and in special hardware for executing them on. Many functional language implementations are based on a system called *graph reduction*. This paper gives an introduction to graph reduction in general, describes some particular schemes used in the past, and then introduces a new scheme which is potentially more efficient. Consideration of the new scheme clarifies the relationships between all the various schemes.

Graph Reduction

A graph reduction *(GR)* computer does not run a program in the conventional sense. Instead, it operates on an expression, continually simplifying it until it is in the simplest possible form. This process of simplification is called *reduction*. If the expression given to the machine is a program in a functional language, then the final result will be the value of the program.

In fact the language in which the expressions within the machine are represented is unlikely to be useable as a functional language, because these should be designed for the convenience of people, not machines. The design of a suitable machine language and the translation into it of user programs is the topic of this paper.

The simplest machine language to be considered is the language of *constant applicative forms (cafs)*. The syntax of this language is

$$\mathbf{E} \; ::= \; \mathbf{C} \; \mid \; (\mathbf{E}_1 \; \mathbf{E}_2)$$

C is a set of *constants* such as integers, characters and booleans. There is a class of constant monadic functions called *operators*. $(E_1\ E_2)$ represents the *application* of the function E_1 to the argument E_2. The function may be an operator or a compound expression. Expressions in this language are simplified by applying operators to arguments. For example, if *NEG* is the integer negation operator then (*NEG* 2) is a constant applicative form, and may be simplified to the integer ~2.

Within the *GR* machine, constant applicative forms can be represented by pairs linked together in the manner of *LISP* list cells. The head of the pair represents the function and the tail of the pair represents the argument. An expression can thus be represented by a single pointer, which is important because it enables expressions to be substituted in others efficiently.

The syntax for application only allows a function to be applied to a single argument. It might be thought that operators of more than one argument could not be fitted into the scheme. Fortunatoly this is not so. Operators of several arguments may be *curried*, meaning that an operator requiring *n* arguments, when applied to one argument, becomes an operator requiring *n-1* arguments with the value of the first argument bound into it. For example, + is an operator requiring two arguments which adds them together, and `(+ 1)` is an operator requiring one argument which adds one to it. `((+ 1) 2)` represents `(+ 1)` applied to `2` and can be simplified to `3`. The *GR* machine implements the currying of operators by not applying an operator until all its erguments are present.

The syntax given becomes very unwieldy when operators with several arguments are used frequently. To ameliorate this it is convenient to assume that epplication is left-associative and omit breckets that serve only to force association to the left. For example, this allows us to write `((+ 1) 2)` as `(+ 1 2)`, dropping the inner brackets. In future this extended syntax will be used to denote expressions within the machine. It is important to realise that when this is done, no extension of the *representation* language used within the real machine is implied. It is simply a more convenient way of writing it. (Those used to *LISP* should take note that this is the *opposite* convention to that used in *LISP*.)

The *GR* machine simplifies *cafs* by locating the leftmost operator and applying it. If the operator can only be applied to operands in their simplest form then the machine simplifies the arguments before applying the operator. In the example,

```
(× (+ 1 2) (+ 3 4))
```

the machine first locates the × operator. × requires simple integers, so the machine goes on to simplify the arguments. This converts the expression into

```
(× 3 7)
```

Now, the multiplication can be performed and the whole expression becomes

```
21
```

This is in its simplest possible form and the machine halts.

How efficient can such a reduction machine be? Let us consider each step of the reduction process with the proposed representation in mind. The leftmost operator can be located by following function pointers until an operator is found. The arguments of an operator are near it in the expression and can be noted at the same time as the operator is being found. Replacing a sub-expression by a simplified version involves only updating a pointer to point to the new version. None of these operations is inherently expensive and so the potential for an efficient *GR* machine seems good.

Implementing Conditionals

So far we have shown how *cats* can be used to represent arithmetic expressions. They will be useless as a machine-code, though, unless they can encode conditionals. One might consider extending the language with the production

$$\mathbf{E} \ ::= \ \mathbf{if}\ \mathbf{E}_1\ \mathbf{then}\ \mathbf{E}_2\ \mathbf{else}\ \mathbf{E}_3$$

but, in fact, this is not necessary. A new operator, *IF*, can be introduced and defined by *reduction rules:*

$$\begin{aligned} &\mathit{IF}\ \mathbf{true}\ \ \mathbf{E}_{\mathbf{then}}\ \mathbf{E}_{\mathbf{else}} \ \rightarrow\ \mathbf{E}_{\mathbf{then}} \\ &\mathit{IF}\ \mathbf{false}\ \mathbf{E}_{\mathbf{then}}\ \mathbf{E}_{\mathbf{else}} \ \rightarrow\ \mathbf{E}_{\mathbf{else}} \end{aligned}$$

The meaning of these rules is that when the machine encounters an expression of the form of the left-hand side, it replaces it by the expression on the right. *IF* requires its first argument to be simplified, then selects one of its other two arguments according to the value of the first. So for example,

$$\mathit{IF}\ ((\ \mathbf{E}\ 0)\ (\mathit{NEG}\ \mathbf{E})\ \mathbf{E}$$

simplifies to the absolute value of E. Conditional expressions in the source language can be implemented by the translation

$$\mathbf{if}\ \mathbf{E}_1\ \mathbf{then}\ \mathbf{E}_2\ \mathbf{else}\ \mathbf{E}_3 \ \Rightarrow\ \mathit{IF}\ \mathbf{E}_1\ \mathbf{E}_2\ \mathbf{E}_3$$

Here and elsewhere the double arrow (⟹) means that the equation is a transformation rule that a compiler would apply, replacing expressions of the form of the left-hand side with the corresponding expression on the right.

Implementing Functions

Even with conditional expressions, *cafs* still seem inedequate as a machine language because there is nothing to correspond to functions in the source language. A new piece of syntax brings them very much closer: the λ-expression [Curry and Feys]. Let us extend the language with the productions

```
E  ::=  V  |  λV. E
```

V is a class of *variables*. λV. E represents a monadic function, which when applied to an argument returns E with the argument substituted for all occurences of V within it. Occurrences of variables in expressions must be *bound* by an enclosing λ-expression. Expressions involving no λs are called *applicative forms*. A new reduction rule is necessary if the machine is to simplify λ-expressions. The rule is to treat λ-expressions as operators with one argument. The result of applying e λ-expression to an argument is obtained by substituting the argument for the bound variable in its body, otherwise called *binding* the argument to the bound variable. For example,

```
(λx. + x 2) 3
```

may be reduced to

```
+ 3 2
```

and thence to

```
5
```

Less trivially,

```
(λy. + y ((λz. × z z) (× y y))) 4
```

may be simplified in the following stages:

```
(λy. + y ((λz. × z z) (× y y))) 4
    + 4 ((λz. × z z) (× 4 4))
      + 4 (× (× 4 4) (× 4 4))
            + 4 (× 16 16)
               + 4 256
                 260
```

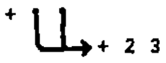
There are two important points to note about this reduction rule. Firstly, the argument of a λ-expression *does not* have to be simplified before substitution -- indeed, if the λ-expression does not use the value of its parameter in delivering its own value, then the argument will *never* be simplified. Secondly, even though substituting an argument may cause it to appear more than once in the result, this does not cause it to be copied or reduced more than once. This is because it is a *pointer* to the argument that is actually substituted. Also, whenever an expression is simplified, the old expression is *overwritten* with the new version. This ensures that all pointers to the old expression now point to the simplified version. To show

```
(λx. + x x) (+ 2 3)
```

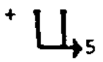
being reduced to

```
+ (+ 2 3) (+ 2 3)
```

is slightly misleading. Rather it is reduced to



Now it is clear that when (+ 2 3) is simplified for the first operand of + the expression becomes



and no second evaluation of (+ 2 3) will be required. This scheme for reducing each expression at most once gives rise to *lazy evaluation* [Henderson & Morris].

Implementing Declarations and Recursion

The λ-notation allows us to write non-recursive functions, but our language is still apparently not very rich. In fact, any functional programming language construct can be translated into the λ-notation. Declarations, both recursive and non-recursive, are the only important constructs that are missing. We shall show how to express these in the λ-notation.

Non-recursive declarations can be implemented by the translation

$$\mathtt{let}\ V = E_1\ \mathtt{in}\ E_2 \Rightarrow (\lambda V.\ E_2)\ E_1$$

As the former should be evaluated by substituting E_1 for V in E_2 it is clear that this translation is correct. Recursive declarations are a little harder, requiring a new basic operator, *Y*. *Y* must be applied to a function, and computes its least fixed point. *Y* is defined by the reduction rule

$$Y\ f \rightarrow f\ (Y\ f)$$

Now, if we translate

$$\mathtt{letrec}\ V = E_1\ \mathtt{in}\ E_2$$

which binds E_1 to V within both E_1 and E_2 into

$$(\lambda V.\ E_2)\ (Y\ (\lambda V.\ E_1))$$

then V acquires the value, within E_2 of

$$Y\ (\lambda V.\ E_1)$$

But, by the rule for Y this is the same as

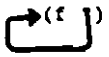
$$(\lambda V.\ E_1)\ (Y\ (\lambda V.\ E_1))$$

which is E_1 with V taking the same value as in E_2. So as V takes the same value in E_1 and E_2 the translation is correct.

Notice that the right-hand side of the rule for Y contains another application of Y. If this application is later reduced the expression becomes `f (f (Y f))`, which can be reduced again to `f (f (f (Y f)))`, and so on. The original expression is clearly equivalent to the infinite application `f (f (f (f ...)))`. A real GR machine would take advantage of the fact that `(Y f)` appears on the right hand side of the rule for Y to build a circular structure. So

`(Y f)`

is actually reduced to



which has the desired effect.

Since we can translate functions and declarations which are the meat of functional languages into the λ-notation, it is unnecessary to extend the machine language any further.

Translating to Combinators

However, there are good reasons for believing that the machine language has already been extended too far. Introducing new syntax with a new reduction rule must inevitably complicate the machine executing the language. For example, the linked pair representation described above is inadequate for the extended language. Also, because λ-reduction may substitute inside the body of a nested λ-expression, the internal structure of these pseudo-operators is important – they are not "pure code". This makes it difficult to use unorthodox representations for λ-expressions, such as microcode. Moreover, there is a danger of inefficiency in substituting into very large λ-expressions, because substitution must visit every leaf of the body. For these reasons, it would be nice to retain the language of constant applicative forms if at all possible.

It should be mentioned that the problems described in the paragraph above can be approached by a scheme of *delayed substitution* [Landin]. In this approach expressions are accompanied by a list of substitutions to be applied to them (the *environment*). The substitutions are only actually performed when simplifying a variable. While this approach is apparently more efficient, Turner has shown that the overhead of manipulating and searching environments may cancel out any gains made elsewhere. Another disadvantage is that, as an expression is no longer complete in itself, but requires an environment for its interpretation, a simplified expression cannot be overwritten with the new version as was described above for *cafs*.

Fortunately, the language of *cafs* can be given the same power as the λ-notation simply by adding a few new primitive operators known as *combinators* [Curry and Feys]. The combinators required are *S*, *K*, and *I*, defined by the rules

$$\begin{aligned} S\ x\ y\ z &\rightarrow (x\ z)\ (y\ z) \\ K\ x\ y &\rightarrow x \\ I\ x &\rightarrow x \end{aligned}$$

To show that the inclusion of these combinators gives the power of the λ-notation it is necessary to define a translation scheme between the two. It is only necessary to define the translation of a λ-expression whose body is already an applicative form, because as the translation produces an applicative form from a λ-expression one may work outwards from the innermost λ-expression eliminating each λ in turn. The translation rules are given by cases of the syntax. If the body is a constant, then

$$\lambda V.\ C \Rightarrow K\ C$$

because of the reduction rule

$$K\ C\ E \rightarrow C$$

For the same reason, if the body is a variable different from the λ-variable then

$$\lambda V_1.\ V_2 \Rightarrow K\ V_2$$

If the body is the bound variable then

$$\lambda V.\ V \Rightarrow I$$

because

$$I\ E \rightarrow E$$

Comparison with the reduction rules shows that each translation rule preserves the meaning of the expression. The only case remaining is that of a λ-expression whose body is an application.

$$\lambda V.\ (E_1\ E_2)$$

But consider the expression

$$S\ (\lambda V.\ E_1)\ (\lambda V.\ E_2)$$

When applied to an expression E_v, this reduces to

$$S\ (\lambda V.\ E_1)\ (\lambda V.\ E_2)\ E_v \ \rightarrow\ ((\lambda V.\ E_1)\ E_v)\ ((\lambda V.\ E_2)\ E_v)$$

that is, E_1 with V replaced by E_v applied to E_2 with V replaced by E_v. But this is exactly the same as $(E_1\ E_2)$ with V replaced by E_v. So

$$\lambda V.\ E_1\ E_2 \ \Rightarrow\ S\ (\lambda V.\ E_1)\ (\lambda V.\ E_2)$$

As this rule produces smaller λ-expressions then the original, repeated application of all four rules must eventually terminate giving an applicative form. Since a program can contain no free variables, it will still contain no free variables after repeated application of the rules, and hence no variables at all. So a program may be translated into a *constant* applicative form. Hence the language of *cafs* already has all the power of the λ-notation.

Turner's Optimisations

This does not mean that there are no further problems involved in using combinators to implement functional languages. In fact, using the translation scheme described above causes programs to grow enormously when they are translated. The reason may be seen by considering the expression

$$(E_1\ E_2)$$

Suppose this expression is enclosed in a λ-expression binding V. Then at some stage it will be translated into

$$((S\ (\lambda V.\ E_1))\ (\lambda V.\ E_2))$$

which requires two cells for its representation, double the number required by the initial version. If it is also contained in a λ-expression binding W, then it will later be translated into

$$((S\ ((S\ (K\ S))\ (\lambda W \lambda V.\ E_1)))\ (\lambda W \lambda V.\ E_2))$$

which requires four cells. In general, an expression contained within *n* nested λ-expressions will expand by a factor of 2^n during this translation.

This would be a prohibitive impediment to using combinators in a real implementation had Turner not shown that the problem may be avoided by introducing a few new combinators and some optimisation rules [Turner]. The simplest and most important of his optimisation rules is the following:

$$S\ (K\ X)\ (K\ Y) \ \Rightarrow\ K\ (X\ Y)$$

justified because

$$\begin{aligned} S\ (K\ X)\ (K\ Y)\ Z &\rightarrow (K\ X\ Z)\ (K\ Y\ Z) \\ &\rightarrow X\ Y \\ &\rightarrow K\ (X\ Y)\ Z \end{aligned}$$

Note that this rule replaces an expression requiring four cells for its representation by one requiring only two. Because of the large multiplying factor described above, this small reduction may lead to a large reduction in the size of the final program. Note that *K*s are "floated" up through the expression as far as possible by this rule. Turner goes on to make use of this by defining new combinators to abbreviate commonly occurring forms. For example, two new combinators *B* and *C* are introduced so that

$$\begin{aligned} S\ (K\ X)\ Y &\Rightarrow B\ X\ Y \\ S\ X\ (K\ Y) &\Rightarrow C\ X\ Y \end{aligned}$$

These may be used as defining properties to deduce reduction rules for *B* and *C*, namely

```
B X Y Z  →  S (K X) Y Z  →  K X Z (Y Z)  →  X (Y Z)
C X Y Z  →  S X (K Y) Z  →  X Z (K Y Z)  →  X Z Y
```

With a few more combinators and reduction rules defined in the same style [Turner], one can prove that translation to combinators does not more than square the size of an expression, and in practise will be considerably better. Indeed, Turner found that the combinator version was often smaller than the original.

These optimisation rules also give an unexpected additional advantage. Because of the first rule, applications of *K* always float out of constant expressions. For example,

```
(+ 1 2)
```

inside a function will be translated into

```
K (+ 1 2)
```

in which the constant expression is still intact. Because it is present in the program from the start of the execution, not built each time the function containing it is called, the expression will be evaluated only once. It will then be overwritten with the value 3 which will be used thereafter. So "constant folding" happens automatically under this implementation.

Moreover, any sub-expression of a λ-expression independent of the bound variable benefits similarly. Consider for example

```
λm. + m (× 2 n)
```

Using the optimisation rules given above, this is translated into

```
C (B + I) (× 2 n)
```

Notice that, just as in the case of the constant expression above, the expression (× 2 n) is preserved intact in the translation. If n is bound to the value 4, for example, then when (× 2 n) is reduced on the first call of the function, its value, 8, will overwrite the original expression, so that the function actually becomes

C (*B* + *I*) 8

In effect, when n is bound to a value the subexpression (× 2 n) becomes a constent expression and so benefits from the automatic constant folding. This property gives moveout from functions, and hence moveout from loops because a loop is just a recursive function. This kind of evaluation scheme is called a *fully lazy evaluation*, because it ensures that each expression is evaluated only once. In contrast, ordinary lazy evaluation ensures that each expression *bound to a variable* is evaluated only once.

With these optimisations graph reduction becomes a practical implementation technique, with the important advantage of full laziness. The disadvantages of Turner's combinator scheme though, are firstly that the machine code form of a program is far removed from the source form. This makes interpretation of intermediate values difficult, for example during debugging. Secondly, compilation is slow. This is partly because of the number of optimisation rules that must be applied. More seriously, the compilation algorithm I have described makes as many passes over each expression as there are λs enclosing it, so compile time is not linear in program size. 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. Referring back to the definitions of *S*, *K*, *I* etc. we see that each one could have been defined as a λ-expression:

$$S = \lambda x \lambda y \lambda z.\ (x\ z)\ (y\ z)$$
$$K = \lambda x \lambda y.\ x$$
$$I = \lambda x.\ x$$

These λ-expressions have two special properties that make them suitable for use as operators. 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*. This means that when *cafs* are substituted for their bound variables, the result is a *caf*. If they are to be used as operators in a *caf* 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 *caf* reduction machine. Where it is necessary to distinguish generalised combinators from Turner's, they are called *super-combinators.*

As there are infinitely many possible combinators, the *GR* mechine 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 aquations, but in practice would probably be compiled into something else, for example microcode. Once again, a translation scheme from general λ-expressions into applicative forms must be provided. This can be done very simply. Take any λ-expression,

$$\lambda V.\ E$$

First the body is converted into an applicative form by invoking the compiler recursively,

$$\lambda V.\ E'$$

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 (λelλnλs. IF (= n 1) (hd s) (el (- n 1) (tl s)))
```

Consider first the innermost λ-expression.

```
λs. 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

```
α n el s  →  IF (= n 1) (hd s) (el (- n 1) (tl s))
```

Now the whole λ-expression can be replaced by

```
α n el
```

and so

```
el = Y (λelλn. α n el)
```

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

```
β el n  →  α n el
γ el    →  β el
```

and so finally

```
el = Y γ
```

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 = λs. IF (= 1 2) (hd s) (el (- 2 1) (tl s))
```

This is equivalent to

```
el 2 = λs. IF-FALSE (hd s) (el 1 (tl s))
```

where *IF-FALSE* is defined by

$$IF\text{-}FALSE\ E_{true}\ E_{false} \rightarrow E_{false}$$

Of course, in a fully lazy implementation, this is what (el 2) would become after one application. This would ensure that the expressions *UF* (= 1 2)) and (el (- 2 1)) are evaluated only once. However, applying the combinators derived above we find

```
el 2  →  Y γ 2
      →  γ el 2
      →  β el 2
      →  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,

```
        λe. IF (= n 1) (hd s) (el (- n 1) (tl e))
```

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

```
        α p q e  →  p (hd s) (q (tl s))
```

and the definition of `el` becomes

```
        el  =  Y (λelλn. α (IF (= n 1)) (el (- n 1)))
```

Continuing the process, β and γ are defined by

```
        β el n  →  α (IF (= n 1)) (el (- n 1))
        γ el    →  β el
```

and `el` is given by

```
        el  =  Y γ
```

as before

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

```
        el 2  →  Y γ 2
              →  γ el 2
              →  β el 2
              →  α (IF (= 2 1)) (el (- 2 1))
```

Now whenever (e1 2) is used, the same copies of the free expressions are used and hence they are evaluated only once. In fact,

$$\text{e1 2} \longrightarrow \alpha\ \textit{IF-FALSE}\ (\alpha\ \textit{IF-TRUE}\ (\text{e1}\ (-\ 1\ 1)))$$

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 free 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

```
γ el  →  β 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 γ been defined by, say,

```
γ n el  →  β (× n 2) el
```

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

```
γ n  →  β (× 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.

```
β el n    →  α n el
α n el s  →  ...
```

In this case no parameters appear to be redundant. However, n and el 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

```
β el n    →  α el n
α el n a  →  ...
```

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

```
α (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 `(α (hd s))` and `(tl s)`. However, if it were rearranged as

```
α (hd s) (tl s) n
```

then the only maximal free expression would be `(α (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

```
α n (hd s) (tl s)
```

making `(α n)` the only maximal free expression. So, to maximise the size and minimise the number of *mfes* of the next enclosing λ-expression, all the *mfes* 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

$$\alpha\ E_1\ \ldots\ E_n$$

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

$$\alpha\ E_1\ \ldots\ E_j$$

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 mfes* in the same manner all the E_i which are *mfes* 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 parameter E_j encloses the native λ-expression of E_i then E_j 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

$$\beta\ p\ q\ r\ s\ \rightarrow\ \alpha\ \ldots\ s\ \ldots$$

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

$$\alpha\ E_1\ \ldots\ E_n\ s$$

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 ($\alpha\ E_1\ \ldots\ E_n$). So in fact, if there are any E_i then β must have been defined by

$$\beta\ p\ s \rightarrow p\ s$$

where p corresponds to ($\alpha\ E_1\ \ldots\ E_n$). 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 ($\beta\ (\alpha\ E_1\ \ldots\ E_n)$) *ie* by $I\ (\alpha\ E_1\ \ldots\ E_n)$). β might as well be eliminated entirely and the λ-expression replaced by ($\alpha\ E_1\ \ldots\ E_n$) 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 (λelλnλs. IF (= n 1) (hd s) (el (- n 1) (tl s)))
```

The innermost λ-expression has two *mfes* (*IF* (= n 1)) and (el (- 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

```
α p q s  →  p (hd s) (q (tl s))
```

and so

```
el  =  Y (λelλn. α (IF (= n 1)) (el (- n 1)))
```

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

```
β el n  →  α (IF (= n 1)) (el (- n 1))
```

making

```
el  =  Y (λel. β el)
```

γ would be defined by

```
γ el  →  β el
```

and so as γ is equal to β it will not be generated. The final result is

```
el  =  Y β
```

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 wall 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-commutetive operator so that parameters of these can be ordered optimally too.

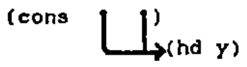
The second is suggested by the observation that application of a combinator involves constructing a tree incorporating the parameter values. There is no reason why a more general graph should not be constructed instead. The difference between a graph and a tree is that some parts may be shared between 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. For exemple,

let x = hd y **in** cone x x

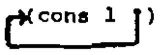
denotes the graph



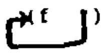
and the notation

let x = cons 1 x **in** x

denotes the graph



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 *cafs* 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.

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.

[Landin] P.J.Landin: "The Mechanical Evaluation of Expressions".
Computer Journal. January. 1964.

[Turner] D.A.Turner: "A New Implementation Technique for Applicative
Languages". Software. Practice and Experience. Vol. 9. 1979.
D.A.Turner: "Another Algorithm for Bracket Abstraction".
The Journal of Symbolic Logic. Vol. 44. No. 2. June. 1979.

Appendix

This appendix contains the experimental results summarised above. Ten small programs were compared and measurements made of the compile time, program size, number of reductions, storage use and run time. The super-combinator implementation appeared to be more efficient for larger programs. As real programs would be far larger than those tested here the super-combinator implementation method should work still better for them.

Table I. Purpose and size of program source in list cells.

Program	Purpose	Size
1	call "twice" (defined in [Turner])	26
2	ackerman's function (curried)	36
3	towers of hanoi	49
4	ackerman's function (non-curried)	51
5	factorial	75
6	test append	93
7	compute 20 primes	106
8	eratosthenes' sieve	115
9	unification algorithm	307
10	compute e to 20 places	317

Table II. Compile-time in seconds.

Program	Size	Old	New	% Gain
1	26	124	177	-43
2	36	166	206	-24
3	49	225	261	-16
4	51	199	243	-22
5	75	230	342	-49
6	93	321	372	-16
7	106	422	429	-2
8	115	463	468	-1
9	307	1591	1341	+16
10	317	2216	1265	+43

Table III. Code size in cells.

Program	Size	Old	New	% Gain
1	26	22	30	-36
2	36	48	43	+10
3	49	70	70	0
4	51	76	62	+5
5	75	91	99	-9
6	93	130	118	+9
7	106	160	145	+9
8	115	176	153	+13
9	307	479	445	+7
10	317	639	435	+31

Table IV. Number of reductions.

Program	Size	Old	New	% Gain
1	26	120	104	+13
2	36	782	410	+48
3	49	2430	1423	+42
4	51	1566	913	+47
5	75	1145	692	+36
6	93	215	126	+42
7	106	7463	5429	+23
8	115	11919	8707	+30
9	307	2713	1675	+39
10	317	257590	103151	+60

Table V. Total cells claimed.

Program	Size	Old	New	% Gain
1	26	65	90	-39
2	36	851	1235	-46
3	49	3300	3626	-10
4	51	1446	1897	-32
5	75	887	967	-9
6	93	199	168	+16
7	106	7463	6108	+19
8	115	8337	7728	+8
9	307	2787	3363	-21
10	317	272208	177712	+35

Table VI. Run time in seconds.

Program	Size	Old	New	% Gain
1	26	0	0	0
2	36	2	2	0
3	49	8	7	+12
4	51	2	2	0
5	75	3	3	+3
6	93	1	1	0
7	106	14	11	+21
8	115	18	16	+11
9	307	6	5	+17
10	317	544	299	+45