8277 COPY 2

THE DESIGN AND IMPLEMENTATION OF PROGRAMMING LANGUAGES

John Hughes

Technical Monograph PRG-40

July 1983
(published as a monograph September 1984)

Oxford University Computing Laboratory
Programming Research Group
8-11 Keble Road
Oxford OX1 3QD

Oxford University Computing Laboratory
Programming Research Group
8 - 11 Keble Road
Oxford OX1 3QD

A thesis submitted for the degree of
Doctor of Philosophy in the University of Oxford,
July 1983

Abstract

The Design and Implementation of Programming Languages

R. J. M. Hughes

Implementation strategies for purely functional languages are reviewed and a new one using "super-combinators" proposed. An efficient algorithm for compilation to super-combinators is described, and realisations of the algorithm are presented in imperative, functional and logic programming languages. The new method is compared with Turner's combinators by an experimental comparison and by a theoretical analysis.

The observed inability of functional programs to make efficient use of store is investigated, and it is shown that this is due to the sequential nature of the underlying abstract machines. Language extensions to incorporate parallelism are introduced and their adequacy is demonstrated in several examples.

Garbage collection is discussed and reference counting selected as the most promising strategy. An extension to reference counting to enable it to collect circular structures is described.

Acknowledgements

I am deeply indebted to Bernard Sufrin, my supervisor. Without his constant encouregement this thesis would never have been written. I am also very grateful to Peter Henderson and Tony Hoare: each of them has influenced my work profoundly. Chris Dollin first interested me in functional languages, and David Wise and Jack Dennis between them interested me in reference counting garbage collection. Geraint Jones listened patiently to many of my ideas before they were even half-baked, and I have had many interesting and enjoyable discussions with Simon Jones and David Turner. Many thanks to them all. Finally, I am very grateful to my wife, Mary, and my parents for their constant moral support, and the Science and Engineering Research Council of Great Britain for their financial support.

THE DESIGN AND IMPLEMENTATION OF PROGRAMMING LANGUAGES

CHAPTER 1

INTRODUCTION

The title of this thesis, "The Design and Implementation of Programming Languages", was chosen before any of the work reported was envisaged to satisfy University regulations. Nevertheless, it is an apt, if cheeky, description of the contents. It is cheeky because we are actually concerned only with functional programming languages. It is apt because our first advice to a language designer would be to make his language functional. We begin in chapter 2 by justifying this advice.

Thereafter we are concerned mainly with implementation. We have been implementing functional languages on and off since 1979 (and Lisp for even longer) and we believe that this experience has given us a good general understanding of the principles involved. In chapter 3 we explain our viewpoint and introduce some terminology for use later.

We have long been an admirer of Turner's combinator implementation method, and much of this thesis is concerned with our own improvement on it using "super-combinators". Our method is explained and analysed in the following three chapters. First, in chapter 4, we explain the principles of our method and a number of enhancements to it. Here we are only concerned with determining what code a program should be compiled into.

In the next chapter we demonstrate that an efficient compiler can be written to generate this code by exhibiting three different ones, written in the imperative, functional and logic programming styles. This also provides an opportunity to compare the three different kinds of language in action. Finally, in chapter 6 we evaluate our new method by a theoretical comparison of its efficiency with Turner's method.

We have observed experimentally that functional programs sometimes require unreasonably large amounts of storage, and that even quite simple programs can gradually clog the memory with useless garbage. We argue in chapter 7 that there are deep-seated reasons for this behaviour, and that it can only be avoided by a fundamental change to parallel abstract machines. We propose an extension to functional languages to control parallelism and justify our choice by showing how it can be used in several examples.

Finally, in chapter 8 we turn our attention to garbage collection. We believe that there are compelling reasons for using reference counting garbage collection, but hitherto this has been awkward because reference counting garbage collectors had difficulty with circular structures. We propose an extension that enables reference counting to be used with any kind of structure.

This completes an overview of our thesis. We wish to note one other point. The reader will find that the words "theorem" and "proof" occur very rarely, if at all. We make no apologies for this. On the contrary, we are following normal mathematical practice in preferring a convincing argument to a formal demonstration. We hope that the reader will gain a better understanding from our informal explanations and "proofs by example" than he would have done from the pages of symbols they replace.

CHAPTER 2

FUNCTIONAL LANGUAGES

2.1. INTRODUCTION

It is often claimed that functional programming will revolutionise the software industry by making programs an order of magnitude easier to write, and yet functional programming is usually defined as "programming without assignment". It is very difficult to see why omitting the assignment statements from one's programs will bring such benefits, so we feel it is worthwhile to examine this point in more detail. We shall try to answer the question "why is functional programming so good".

It is helpful to recall the history of another software revolution: structured programming. Very similar claims were made for structured programming in its early days, and yet it was often defined as "programming without the goto". It is difficult to see why avoiding goto statements should make programming easier. In fact, of course, this negative definition of structured programming betrays a gross misunderstanding of the whole idea. It is not the omission of gotos that makes structured programming easier, it is the inclusion of new structured programming constructs such as the while loop and if-then-else in new languages like Pascal. One can omit gotos in FORTRAN IV until one is blue in the face without making programming any easier.

In the same way the negative definition fails totally to capture the spirit of functional programming. Its success is not due to the omission of assignment, but to the inclusion of new, powerful features in new functional programming languages. It would even be possible to design a functional programming language including assignment, in the same way as Pascal includes the goto; but, like the Pascal goto, assignment would be neither useful nor necessary. In section 2 of this chapter we shall endeavour to identify the important features of functional programming languages. Some of these features are also found in non-functional languages such as Gedanken [Reynolds70], Scheme [Steele78] and ML [Gordon79]. We do not feel that this weakens our case at all. In fact, it strengthens it, because each of these languages is nice precisely because it includes some of the features we are applauding. None of them includes all the important features we will describe, however.

Another similarity between structured and functional programming is that programs have nicer formal properties than their unstructured or non-functional counterparts (single entry and exit of blocks in structured programs, and referential transparency in functional programs). This is important, but we do not feel that it is a major factor in making programs easier to write. Rather, it opens up the possibility of formal program development. In section 3 we discuss the close relationship between functional programming and formal specification as it is practised at Oxford, and describe our view of how programs could be developed in the future.

2.2. FUNCTIONAL FEATURES

We have said that we believe functional languages make programming easier because they include new, powerful features. But what exactly are these features? In this section we identify those we feel are the most important.

One reason why the functional programmer does less work is that the functional system does more work. The system assumes responsibility in a number of areas, freeing the programmer from having to think about them at all. One important area is storage allocation. The functional programmer is not required to decide the lifetimes of the objects he creates; instead the system deletes them and reuses the space they occupied when there are no references to them left. This avoids the risk of bugs caused by deleting objects too early, or excessive storage use caused by deleting them too late (a "dumb" garbage collector or bad virtual machine design can still cause objects to be deleted too late). The most important advantage, however, is that the programmer does not need to think about it.

Another area in which the system assumes responsibility is that of deciding evaluation order. In imperative languages, the evaluation order is explicit in the program, but in functional languages the system is free to use any suitable order; this gives rise to the possibility of several different strategies for determining order, including lazy evaluation. This is even more important than automatic storage allocation, because it means that the structure of a program need no longer be determined by the desired evaluation order. Great simplifications can result: for example, problems that are usually solved by backtracking to find the first solution can instead be solved by writing functions that return a list of all solutions. If only the first solution in the list is used, then only that solution will be computed; on the other hand, if that solution proves unsatisfactory and the next solution is used instead then the system will pick up where it left off (that is, we can get backtracking behaviour without writing backtracking programs). Many other problems can be solved most naturally by programs whose structure does not reflect the desired evaluation order: for a larger example, see section 5.4.

Another consequence of breaking this connection is that we can program using infinite data-structures, provided we never require the system to compute all their components. Non-terminating programs can often be expressed very neatly as functions on infinite data-structures. Their use can also improve the modularity of terminating programs: for example, a numerical iteration can be expressed as one function that computes an infinite list of approximations and error bounds, and another that selects a particular element of that list. This separates the concerns of computing the approximations, and deciding, using whatever criterion seems appropriate, which approximation to settle for.

Infinite data-structures are often applied to input and output. Typically a program is passed an infinite list of inputs and returns an infinite list of outputs, and the system chooses an execution order in which the outputs are computed as the inputs are consumed. Programming in terms of the entire input or output of a program allows programs to be combined very easily: for example, functional operating systems will not need special "pipes" in order for one program to read another's output.

Turning to more specific features, higher-order functions are one of the most valuable. These are functions that take other functions as arguments or return them as results. Using higher-order functions we can effectively extend our language with whatever control structures we desire. We can use this to improve the modularity of our programs by introducing special purpose control structures as part of abstract types (cf. iterators in CLU [Liskov79]). For example, the well known map function is an appropriate control structure for list processing. It enables us to write certain functions on lists without knowing the details of their internal structure.

The recursion equation style of function definition is also very valuable. In essence this is just another way of writing certain conditionals. Its advantages are that equations are usually more readable than conditionals, and that it allows quite complex conditions to be expressed very simply. This is particularly true of functions of many arguments, where a sequence of tests can become very unperspicuous.

The expressive power of functional languages is considerably increased by incorporating a "set-expression" notation, as Turner does in KRC [Turner81]. This notation seems ideal as a functional iteration construct. It brings the same clarity to functional programs as the for-loop brought to imperative programs.

Finally, a valuable feature which is not confined to functional languages, but which fits easily into their conceptual framework, is the provision of abstract data-types. Implementation is easy: all that is necessary is an abstraction function that stamps objects with a type, its inverse, and a type testing function. This simple extension allows the programmer to make his intention far clearer, and allows the system to catch far more errors. (With this simple kind of abstract data-type the programmer is able to express the conceptual difference between two objects with the same representation by stamping them with different types, and so the system is able to check that the usage of these objects is consistent with the programmer's intention. Careful use of scope is necessary to guarantee that only a particular set of operations is applied to objects of a particular type, so this is not quite an implementation of "abstract data-types" as the term is usually understood).

We have described the particular features we feel are most important in making functional languages powerful and easy to use. Equally important, however, is the fact that the various language elements can be combined in any way. There is no host of special cases and restrictions in a functional language, in contrast to Fortran, Pascal and Ada: each construct may be described simply and used wherever the programmer chooses. This simplicity and freedom allows one to become conversant with the language quickly, and to use it confidently.

2.3. FORMAL SPECIFICATION

A very compelling reason for favouring functional programming is that it fits very well with formal program development. We shall illustrate this by describing how programs are developed formally at Oxford and explaining why using functional programming languages can simplify the process greatly.

To begin with, a formal specification of the task to be performed is written. This consists of a number of definitions of types and functions on them. These definitions are constructed in terms of the basic objects of mathematics: sets, natural numbers, relations, etc. Definitions may be constructive, effectively giving a method for computing the object defined, or non-constructive. For example, the squaring function can be defined constructively by

```
square: N → N
square = λn:N. n*n
```

and the square root function can be defined non-constructively by

$$\sqrt{} : \mathbf{N} \twoheadrightarrow \mathbf{N}$$
$$\sqrt{} = \mathtt{square}^{-1}$$

Having written the formal specification, the programmer transforms it into an entirely constructive one, by providing constructive definitions for all his functions and proving the new definitions equivalent to the old ones. At the end of this stage, the specification not only defines what is to be computed, it describes a reasonable way of computing it.

The final stage of program development consists of taking the final specification and translating it into Pascal. The translation is done informally, and is not proved correct. Indeed, the resulting program usually fails to meet the specification because it has implementation limits built into it.

This last stage is a great inconvenience to formal program developers. It cannot be proved correct, because in general it isn't correct. Moreover, it involves a lot of work at a very low level, organising storage management, execution order etc. In fact, it is totally unnecessary. The final specification, being completely constructive, is already a functional program. There is no need to go any further. Stopping at this stage not only eliminates the informal part of program development - it also saves the programmer a lot of work.

To summarise, formal specification offers functional programmers a way of developing provably correct programs. Functional programming offers formal specifiers a way of implementing their specifications correctly and easily. Together, they will be powerful indeed.

CHAPTER 3

IMPLEMENTATIONS

3.1. INTRODUCTION

In this chapter we discuss the main implementation techniques used for functional languages, and investigate their advantages and disadvantages. Beginning with the observation that all functional languages can be translated into the λ-calculus plus constants, we take the λ-calculus as the canonical functional language. After a description of reduction, which is an execution paradigm appropriate to the λ-calculus, we go on to develop specific implementations. We consider machine representations, and explain why we choose a graph. We identify the major inefficiency of a graph-reduction λ-reducer, and show how different attempts to cure it can lead to the SECD machine and Turner's combinators.

3.2. THE λ-CALCULUS

We begin with a brief description of the λ-calculus [Curry58], adopting the following abstract syntax for expressions:

$$\mathbf{E} \;::=\; \mathbf{C} \mid \mathbf{V} \mid (\mathbf{E}\ \mathbf{E}) \mid \lambda\mathbf{V}.\mathbf{E}$$

where **C** ranges over a set of constants and **V** ranges over a set of variable names. (E_1 E_2) represents E_1 applied to E_2 and λV.E represents the function of V that E is. C includes numbers, booleans etc., and also basic functions such as cons. λV.E is said to bind the variable V inside the body, and unbound variables are said to be free.

The meaning of an expression is defined by reduction rules of the form E_1 **red** E_2, which means that wherever an expression of the form E_1 appears, it may be replaced by E_2. Each basic function has its own rule, for example

```
        head (cons a b) red a
        + m n red m+n
```

provided that m and n are natural numbers. (Here we adopt the convention that application is left associative and drop unnecessary brackets). The effect of applying a λ-expression is defined by the β-rule

(β)   (λV.E) E_0 **red** E[E_0/V]

where the right hand side stands for E with all free occurrences of V replaced by E_0.

However, the β-rule is only applicable when E_0 has no free variables. This is more restrictive than the β-rule as usually stated, but we don't care because we are interested in reducing complete programs, which *a priori* have no free variables. This means that the top-level application of a program must have arguments with no free variables either, and so the restricted β-rule is applicable. If the top-level of a program ever becomes a λ-expression, then we are content not to perform any reductions on the body of this λ-exprsssion, because we regard reductions inside functions as program transformation, not program execution. The advantage we gain by this is that we never need to invoke the expensive α-rule (which renames variables to avoid inadvertant binding).

An implementation of the λ-calculus must operate on an expression by applying reduction rules to it until no more are applicable. The expression is then said to be in normal form, and this normal form is the result of the computstion.

The choice of the reductions to be performed at any particular moment is a matter of implementation strategy. It is constrained by the fact that certain functions (for example, +) are strict, thet is can only be applied to arguments in normal form, and by the fact that reductions of the top-level of the program should alweys be performed since they lead most directly to production of the answer. Within these constraints we may choose:

(1) never to reduce an application unless the argument is in normal form. This gives a "strict" or "call-by-value" semantica which we argued against in chapter 2.

(2) to reduce all reducible expressions in parallel. This gives "eager" evaluation, which may waste resources as some reductions might never have been necessary.

(3) to perform only reductions at the top-level of the program, or those directly necessary to enable a strict top-level reduction to take place. This gives "lazy evaluation". This approach can run into some very subtle problems (see chapter 7).

(4) some combination of (2) and (3). This is the approach we advocate, and will elucidate in chapter 7.

In order to run real functional programming languages on a λ-calculus machine, we must translate them into the λ-calculus. We define translation rules of the form E_1 trans E_2, which when applied repeatedly will translate a program into the equivalent λ-expression. A complete set of translation rules would be too long-winded to reproduce here, but we give a few examples:

```
E₁ + E₂ trans + E₁ E₂
let I = E₁ in E₂ trans (λI.E₂) E₁
if E₁ then E₂ else E₃ trans IF E₁ E₂ E₃
```

where IF is defined by

```
IF true E₁ E₂ red E₁
IF false E₁ E₂ red E₂
```

Recursive declarations are translated by the rule

```
letrec I = E₁ in E₂ trans let I = Y (λI.E₁) in E₂
```

using the basic function Y which we have not seen before. The reduction rule for Y is

```
Y f red f (Y f)
```

which uses Y f again on the right hand side. This means that Y f has no normal form, because it reduces to (f (f (f ...))). It is this potentially infinite behaviour that allows Y to implement recursion. Y is very important and we shall return to its implementation in the next section.

Mutual recursion is a little trickier. The syntax we use is

letrec I_1=E_1 **and** I_2=E_2 **and** ... **in** E_0

We need to introduce two new rules to translate it, firstly

I_1=E_1 **and** I_2=E_2 **trans** (I_1.I_2) = cons E_1 E_2

This translates the mutually recursive definitions into a single recursive "structured" definition, which declares several names to be the components of a structure. This single definition can be translated further by our earlier rules, producing a λ-expression binding a "structured variable". We translate this kind of λ-expression into simple λ-expressions as follows:

λ(I_1.I_2)E **trans** U (λI_1λI_2.E)

using an explicit unpacking function U defined by

U f x red f (head x) (tail x)

We hope that these examples have convinced the reader that translation into the λ-calculus is not an onerous task.

3.3. REPRESENTATION

A critical choice in the design of a reduction machine is the choice of the representation of expressions. There are two essentially different possibilities: as a string, or as a graph.

In string reduction machines the program is stored as a sequence of symbols, much as it appears on paper. During function application the machine actually substitutes one expression into another, and must shuffle the symbols around to make room. This is potentially a costly operation. Still worse, if the argument of a function is not in normal form then the substitution may greatly increase the amount of work to be done, since it may create several copies, each of which must be reduced independently. For example, in the case

```
(λx.+ x x) expensive red + expensive expensive
```

then the amount of work is doubled, assuming that reducing "expensive" is considerably more costly than an addition or a function application. for this reason string reduction is really only suitable for strict languages, which guarantee that arguments are in normal form before substitution. Mago's string reduction machine, for example, uses FFP which is indeed a strict language [Mago79]. Since we believe non-strictness to be a vital programming tool we reject string reduction.

In contrast, a graph reduction machine represents every expression as a cell which contains pointers to its sub-expressions. Now when an argument is substituted into a function body it is not necessary to copy the argument: instead a pointer to the argument can be substituted. This makes function application much more efficient. Not only this, it also means that there is only ever one copy of the argument. The first time it is reduced it will be replaced by its normal form, and all pointers to the argument will now automatically refer to its normal form. This means that no matter when the argument is first reduced, it will not be reduced more than once. Graph reduction machines therefore provide good support for non-strict languages.

They also permit an efficiant implementation of Y. Recall that the reduction rule for Y is

```
Y f red f (Y f)
```

On a graph reduction machine we can take advantage of the fact that a copy of the left hand side appears on the right by constructing a circular result:

$$\begin{tikzcd} Y\ f\ \text{red}\ f & \bullet \arrow[l, bend left] \end{tikzcd}$$

It is clear that this reduction rule is equivalent to the original one, but is much more efficient since it reduces (Y f) completely in only one step.

The disadvantage of graph reduction machines is that, just as there is only ever one copy of an argument, so there is only ever one copy of a function. This means that substitution during function application must not be destructive - for if it were then the function would be corrupted and could not be called again. Therefore, instead of copying the argument when we apply a function, we must copy the function body. This is the fundamental inefficiency of graph-reduction; the techniques we shall discuss in the rest of this chapter seek to eliminate or reduce it.

3.4. THE SECD MACHINE

As we remarked above, in a graph reduction architecture function application requires copying the function body, and this can become very expensive. Consider for example

```
λaλbλc. E
```

When this function is applied, E would have to be copied three times, once to substitute for a, once to substitute for b, and once to substitute for c. In general, an expression would be copied once for every variable in scope in it. Since there are often many variables in scope at a time this is prohibitively expensive.

One of the attempts to avoid this is the SECD machine [Landin64]. The fundamental idea is to delay substitutions until the very last minute, so that when we are finally forced to perform them we can do several together, copying the expression only once. To do this we need two new data types: an "environment", which is a mapping from identifiers to values, and a "closure" which consists of an expression and an environment of delayed substitutions to apply to it. Writing a closure as $E\{\rho\}$, where ρ is the environment, we change the reduction rules of the machine to

$$(E_1\ E_2)\ \{\rho\}\ \mathbf{red}\ E_1\{\rho\}\ E_2\{\rho\}$$
$$((\lambda I.E_1)\{\rho_1\})\ (E_2\{\rho_2\})\ \mathbf{red}\ E_1\{\rho_1 \oplus (I \mapsto E_2\{\rho_2\})\}$$
$$C\ \{\rho\}\ \mathbf{red}\ C$$
$$V\ \{\rho\}\ \mathbf{red}\ \rho(V)$$

It is clear from this description that the SECD machine is still a graph reduction machine. Like other graph reduction approaches, it provides good support for non-strict languages, and for the same reason: the environments contain only pointers to expressions, and so each expression is only reduced once.

However, real SECD machines use four registers, the stack, environment, control and dump to avoid constructing some of the intermediate closures (this is the reason for the name). This optimisation, coupled with the more complex reduction rules, makes the behaviour of an SECD machine quite difficult to reason about. Because of this we prefer to work with simpler approaches. Some of our results cannot be applied to SECD machines at all.

The SECD machine does indeed avoid doing any substitutions, but it replaces them with environment lookups which can prove very expensive when there are many names in the environment. Turner found that his SECD implementation spent most of its time looking up names in the environment [Turner79]. This has led to other attempts to reduce the cost of substitution.

3.5. CAF REDUCTION

Since substitution is the major inefficiency in a graph reduction machine, and substitution only occurs when λ-expressions are applied, it is natural to try to eliminate λ-expressions and variables altogether. To this end we define the language of *constant applicative forms* (cafs) whose syntax is

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

This language is particularly simple and easy to implement; however, for it to be useful we must have a way of translating λ-expressions into it. To do this we have to introduce a new class of constant, the *combinator*. These have the syntax

$$\mathbf{C} \ ::= \ \kappa\ \mathbf{I}_1 \ \ldots\ \mathbf{I}_n .\ \mathbf{B}$$
$$\mathbf{B} \ ::= \ \mathbf{C} \mid \mathbf{I} \mid (\mathbf{B}\ \mathbf{B})$$

The *reduction* rule for combinators is

$$(\kappa \mathbf{I}_1 \ldots \mathbf{I}_n . \mathbf{B})\ \mathbf{E}_1 \ldots \mathbf{E}_n \ \mathbf{red}\ \mathbf{B}[\mathbf{E}_1/\mathbf{I}_1, \ldots, \mathbf{E}_n/\mathbf{I}_n]$$

A combinator is very like a λ-expression, but there are two important differences: firstly, a combinator takes several arguments at once, and secondly, a combinator must have no free variables. Taken together, these mean that an expression can never be substituted into twice, since the only expressions that can be substituted into are combinator bodies, and the result of substitution is not a combinator body. So we have eliminated the major inefficiency of graph reduction.

We may translate the λ-calculus into cafs as follows: if the free variables of λV.E are $I_1 \ldots I_n$, then

$$\lambda V.E \text{ trans } (\kappa I_1 \ldots I_n\ V.E)\ I_1 \ldots I_n$$

Repeated applications of this rule will eliminate all the λ-expressions in the program, and it is clear that the resulting combinators will have no free variables, as required. This approach has been used by Johnsson in his ML compiler [Johnsson83].

3.6. THE SKI MACHINE

It has long been known that λ-expressions can be translated into cafs which use only three different combinators.

$$S = \kappa abc.\ a\ c\ (b\ c)$$
$$K = \kappa ab.\ a$$
$$I = \kappa a.\ a$$

by the translation rules

```
λV.V trans I
λV.V1 trans K V1
λV.E1 E2 trans S (λV.E1) (λV.E2)
```

However, applying these translation rules to an expression of any size yields an enormously large result. Turner observed [Turner79] that using an optimisation rule

```
S (K a) (K b) trans K (a b)
```

drastically reduces the size of the result. He used this rule, together with a few other combinators which abbreviate common forms, to make a practical cal-reduction implementation of SASL. He found that his implementation performed considerably better than his SECD-machine implementation of the same language, apparently because the excessive cost of environment lookups was avoided.

He found that the use of his optimisation rule had another interesting consequence, which we may illustrate by considering the example

```
λx. + 1 2
```

Of course, every time this function is applied it returns the answer 3. We are interested in whether the addition is performed on every call, or once only. Using the λ-reduction rules, we find that

```
(λx. + 1 2) a red + 1 2 red 3
```

The addition is performed on every call, since the function body is copied during application.

Translating to combinators, we get

```
λx. + 1 2 trans S(S(K +)(K 1))(K 2)
```

and when this function is applied, we find

```
S(S(K +)(K 1))(K 2) a red S(K +)(K 1)a(K 2 a)
                      red K + a (K 1 a) 2
                      red + 1 2 red 3
```

So, as in the case of direct λ-reduction, the expression (+ 1 2) is constructed and reduced on every call.

But, using Turner's optimisation rule, we find that

```
λx. + 1 2 trans K (+ 1 2)
```

and

```
K (+ 1 2) a red + 1 2 red 3
```

In this case, the result of applying K (+ 1 2) is a pointer to K's argument, (+ 1 2). When this expression is reduced to 3, K's argument becomes 3. So, after the first call the function is simply (K 3), and it returns 3 without performing any additions at all.

We have just demonstrated that Turner's optimisation rule guarantees that constant expressions are only evaluated once. Its effect is more far-reaching than this, though, because it treats any expression inside a function that does not involve the bound variable in the same way. It guarantees that any such expression is evaluated only once, and that its value will be used directly thereafter. Thus it subsumes such optimisations as constant folding and move-out from loops in imperative languages.

We may summarise our conclusions as

> Every expression is evaluated at most once after the variables in it have been bound.

We call this property *fully lazy evaluation*. It is analogous to lazy evaluation, but is more general since the latter states only that every *argument of a function* is evaluated at most once.

CHAPTER 4

SUPER-COMBINATORS

4.1. INTRODUCTION

We have described two different approaches to improving the efficiency of graph-reduction implementations of the λ-calculus. The combinator technique of section 3.5 is simple, and has the advantage that compilation into those combinators is easy. Turner's method, on the other hand, requires a more complex compiler and breaks the execution down into very small steps: an application of S, for example, does not achieve very much compared to an application of some larger combinators. However, it brings with it the very important advantage of fully lazy evaluation.

In this chapter we shall develop a method that combines the best features of both approaches. Evaluation will be fully lazy, but individual combinators will accomplish much. The basic method will be introduced in section 4.2. Subsequent sections present various improvements that can be made, and the final section presents the results of an experimental comparison of our super-combinators with Turner's approach.

4.2. THE BASIC METHOD

Our purpose is to modify the approach of section 3.5 so that it avoids unnecessary repeated evaluation of expressions that are independent of some bound variables. We first of *all* give these expressions a name: they are the *free expressions* of λ-expressions, by analogy with free variables. In fact, free variables are the *minimal* free expressions of a λ-expression. We call free expressions that are not part of any larger free expression *maximal* free expressions (mfes). For example, the λ-expression λy.(+ (* x x) (* y y)) has many free expressions, including +, *, and (* x x) since none of these expressions involves y, but it has only two maximal free expressions, being (+ (* x x)) and *.

We can now define a new translation scheme. Consider the λ-expression λV.E, and suppose that E_1 ... E_n are its maximal free expressions. Let I_1 ... I_n be identifiers not used in the λ-expression. Then

$$\lambda V.E \ \mathbf{trans}$$
$$(\kappa I_1 \ldots I_n V.E[I_1/E_1, \ldots, I_n/E_n]) \ E_1 \ \ldots \ E_n$$

By the definition of combinator application, it is clear that both sides of this equation are equivalent, and therefore the translation is correct. Also, the combinator we have produced cannot have any free variables, because any free variable is a free expression, and would therefore be enclosed in one of the maximal free expressions E_1 ... E_n. Therefore the translation produces genuine combinators. As in section 3.5, repeated application of the rule will translate a whole program to combinators.

We have yet to show how this method gives fully lazy evaluation. We shall first of all consider our previous example. λx.+ 1 2. This is translated as

```
λx.+ 1 2 trans (κax.a) (+ 1 2)
                      = K (+ 1 2)
```

exactly the same code as produced by Turner's approach. When this function is applied it returns a pointer to the argument. (+ 1 2), and so when this is reduced to 3 the function actually becomes (K 3). No further additions are performed.

To take a more realistic example, we consider the function that selects the nth element of a sequence.

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

This function can be partially parameterised, as in (el 2), to give a function that always selects a particular element of a sequence. Fully lazy evaluation in this case would mean that such a partially parameterised function would need to do no arithmetic on n to select the right element.

Looking at the innermost λ-expression first, we see that it has maximal free expressions (IF (= n 1)) and (el (- n 1)). It is translated into

```
α (IF (= n 1)) (el (- n 1))
where α = κabs. a (hd s) (b (tl s))
```

When we apply the same process to the other λ-expressions too, we end up with

```
el ≙ Y γ
γ ≙ κel. β el
β ≙ κel n. α (IF (= n 1)) (el (- n 1))
α ≙ κa b s. a (hd s) (b (tl s))
```

Now, when the partially parameterised function (el 2) is first applied to a sequence, it will be reduced as follows:

```
el 2 red Y γ 2
     red γ el 2
     red β el 2
     red α (IF (= 2 1)) (el (- 2 1))
     red α (IF false) (el 1)
 ... red α (IF false) (α (IF true) (el 0))
```

All of these reductions will be performed on the first call, transforming (el 2) into a function that selects the second element of a sequence without doing any arithmetic.

Notice that we did *not* bother to abstract out constant free expressions in this example. In fact, it is unnecessary to do so *in order* to guarantee that the combinators generated have no free variables. It is convenient, both on paper and in implementations, to allow combinators to include constant expressions, but with the *understanding* that the full laziness property still applies. We modify our understanding of combinator application so that (κa.+ 1 2) is equivalent to (κab.a) (+ 1 2).

So, this basic method gives us a fully lazy evaluation of our original program, without breaking execution down into small steps – indeed, the code produced contains only one combinator for each source λ-expression. Experiments show that it is already more efficient than Turner's method. In subsequent sections we will see how it can be improved still further.

4.3. PARAMETER ORDER

The basic super-combinator method does not define the combinators uniquely, since we allowed the parameters of the combinator to appear in any order. We should ask ourselves whether one order is likely to be better than another, or whether all orders are equally good.

One factor which might influence a choice of parameter order is the desire to eliminate redundant parameters and combinators. An example of a redundant combinator is $\kappa a.\alpha a$, which is equivalent to α. Using α instead is more efficient, since it eliminates an unnecessary combinator application. Parameter order has a bearing on this in a case such as $\beta \Leftarrow \kappa ab.\alpha ba$, which is not equivalent to α. If α is a combinator introduced by the compiler, and the compiler has the choice of the order in which the parameters of α appear, then it can choose the other order, making $\beta \Leftarrow \kappa ab.\alpha ab$ which is equivalent to α. Thus a correct choice of parameter order can help make more combinators redundant.

An example of a redundant parameter is b in $\kappa ab.\alpha(+\ a\ 1)b$, which is equivalent to $\kappa a.\alpha(+\ a\ 1)$. The advantage of eliminating b in this case is that this will permit the combinator to be applied when fewer arguments are available, and hence will allow the result of the application to be shared more widely. This will reduce the total number of combinator applications necessary. Once again, a correct choice of parameter order for α is essential, since otherwise β would be κab.αb(+ a 1) in which no parameters are redundant.

Another factor influencing choice of parameter order is the desire to make maximal free expressions as large as possible. If, by rearranging the parameters of one combinator, we can make several mfes of an enclosing λ-expression combine into one larger one, then we have improved the efficiency of the enclosing combinator. This is analogous to trying to move as much work as possible out of a loop. For example, consider

```
λn. α (hd s) (+ n 1) (tl s)
```

which has mfes (α (hd s)) and (tl s). This λ-expression will be translated into

```
(κabn. a (+ n 1) b) (α (hd s)) (tl s)
```

However, if the parameters of α had been arranged as follows:

```
λn. α (hd s) (tl s) (+ n 1)
```

then the λ-expression would have been replaced by

```
(κan. a (+ n 1)) (α (hd s) (tl s))
```

The latter form is more efficient, both because the combinator has fewer arguments and a simpler body, and so is more efficient to apply, and because the large expression (α (hd s) (tl s)) need only be constructed once and can then be shared between all calls of the function.

We derive a general rule from this example. The combinator α is derived from a λ-expression with mfes (hd s), (tl s) and (+ n 1). We have seen that, when choosing a parameter order for α, those mfes which are also free expressions of the next enclosing λ-expression should appear before those which are not. Suppose α has parameters E_1 ... E_n, so that the call of α will appear as

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

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 for all k greater than j, E_k is not. This guarantees that (α E_1 ... E_j) is 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 free in it should appear before the E_k which are not, and so on and so forth. In general, the optimal ordering under this criterion can be established as follows. Every E_i is a free expression of one or more enclosing λ-expressions. Call the innermost λ-expression in which E_i is not free its *native* λ-expression. This is the innermost λ-expression which binds a variable in E_i. If the native λ-expression of E_i encloses the native λ-expression of E_j, then E_i precedes E_j in the optimal ordering. This does not define the optimal ordering uniquely, because expressions with the same native λ-expression can appear in any order. This doesn't matter, because any ordering satisfying our condition is as optimal as any other.

Notice that an expression has no meaning outside its native λ-expression, because the bound variable of its native λ-expression appears in it somewhere. Also, constant expressions have no native λ-expressions at all, because they are free in all λ-expressions. For the sake of uniformity they are regarded as being native to some notional λ-expression enclosing the whole program.

We have deduced an optimal ordering to maximise the size of mfes. Let us return to the subject of redundant parameters. First we observe that the compiler can only choose the order of parameters of combinators, and so the only case in which it can help make parameters redundant is when one combinator is defined directly as a call of another. For example, suppose β is defined by

$$\beta = \lambda pqrs.\ \alpha\ \ldots s \ldots$$

No parameters are redundent unless s is, but s was the bound variable of the λ-expression β was derived from. Therefore, 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 simply s, then s is a redundant parameter and can be eliminated; otherwise s is not redundent and nor are any of the other parameters. If s is redundant, then the call of α must take the form

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

where s does not occur in E_1 to E_n. Therefore each E_i is free in β, and hence so is (α E_1 ... E_n). If n is non-zero, then β would actually be defined by

$$\beta = \lambda ps.\ p\ s$$

where p corresponds to (α E_1 ... E_n). If n is zero, then

$$\beta = \kappa s.\ \alpha\ s$$

In the first case β is equal to I and we may eliminate it, using (α E_1 ... E_n) instead of (β (α E_1 ... E_n)). In the second case β is equivalent to α, and so can be eliminated. So we see that the optimal ordering we have defined also guarantees that parameters and combinators will be made redundant if possible, and moreover, detection is reduced to looking for two simple cases.

We shall demonstrate the importance of this optimisation by exhibiting a (rather pathological) example where it makes an enormous difference to the size of code produced. Consider the function F_n that applies a function G to its n arguments in reverse order.

$$F_n = \lambda I_1 \ldots \lambda I_n.\ G\ I_n\ \ldots\ I_1$$

The maximal free expressions of the innermost λ-expression are I_1 ... I_{n-1}, so, if we do not choose the optimal order, we could translate

$$\lambda I_n.\ G\ I_n\ \ldots\ I_1$$

into

$$\alpha_n\ I_{n-1}\ \ldots\ I_1$$
$$\textbf{where}\ \alpha_n = \kappa I_{n-1} \ldots I_1 I_n.\ G\ I_n\ \ldots\ I_1$$

where no parameters are redundant. But now

$$F_n = \lambda I_1 \ldots \lambda I_{n-1}.\ \alpha_n\ I_{n-1}\ \ldots\ I_1$$

which is in the same form as it was in originally. If we continue the translation in this way, we will define $\alpha_{n-1} \ldots \alpha_1$, where

$$\alpha_i \equiv \kappa I_{i-1} \ldots I_1 I_i.\ \alpha_{i+1}\ I_{i-1}\ \ldots\ I_1$$
$$F_n \equiv \alpha_1$$

Here α_i is of size O(i), and so we will translate the λ-expression of size O(n) into code of size $O(n^2)$. This is bad news indeed.

On the other hand, if we use the optimal parameter ordering, we will be forced to define

$$\alpha_n \equiv \kappa I_1 \ldots I_n.\ G\ I_n\ \ldots\ I_1$$

giving

$$F_n \equiv \lambda I_1 \ldots I_{n-1}.\ \alpha_n\ I_1\ \ldots\ I_{n-1}$$

All the other α_i will be radundant, so we will finally define $F_n \equiv \alpha_n$. The code will be of size O(n). The tremendous improvement in this example leads us to expect a significant improvement in practice. In chapter 6 we will show that using the optimal parameter order leads to an improvement in the complexity of the code size.

4.4. OPTIMISING CONDITIONALS

When the method described above is used in practice we find that, on the whole, it performs well, but in some circumstances performs worse than Turner's method. An example where this happens is the naive Fibonacci function

```
fib = Y (λfibλn. IF (< n 2)
                      n
                      (+ (fib (- n 1))
                         (fib (- n 2)))))
```

After translation into super-combinators, fib looks like

```
fib = Y α
α = κf n. IF (< n 2) n
             (+ (f (- n 1)) (f (- n 2)))
```

Because of the divide-and-conquer nature of the function, fib is called with argument 0 or 1 disproportionately more often than with other values. In these cases it is reduced as follows

```
fib 1 red α fib 1
      red IF (< 1 2) 1
             (+ (fib (- 1 1)) (fib (- 1 2)))
      red IF true 1
             (+ (fib (- 1 1)) (fib (- 1 2)))
      red 1
```

and we see that the large expression (+ (fib (- 1 1)) (fib (- 1 2))) is constructed and never evaluated. The cost of constructing this expression is likely to outweigh the rest of the cost of computing (fib 1). Since the expression is never actually required, this effort is entirely wasted.

In fact, this kind of situation can occur whenever we write a conditional expression. In the expression (IF E_1 E_2 E_3), it is certain that only one of E_2 and E_3 will actually be required, and so to construct both is wasteful.

Turner's approach actually avoids constructing the unnecessary one, since the construction is done only on demand. In effect, Turner treets $(\lambda x.\ \mathrm{IF}\ E_1\ E_2\ E_3)$ as $(\lambda x.\ \mathrm{IF}\ E_1\ ((\lambda x.E_2)\ x)\ ((\lambda x.E_3)\ x))$. Since both functions $\lambda x.E_2$ and $\lambda x.E_3$ ere independent of x, the only expressions that need to be constructed on eech call are the applications of these functions to x.

Fortunately, the same technique works for super-combinators. When compiling an expression

$$\mathrm{IF}\ E_1\ E_2\ E_3$$

with the enclosing λ-expression binding x, we can treat it as

$$\mathrm{IF}\ E_1\ ((\lambda x.E_2)x)\ ((\lambda x.E_3)x)$$

In this case $\lambda x.E_2$ and $\lambda x.E_2$ will be free expressions of the enclosing λ-expression, and the combinator body will contain only

$$\mathrm{IF}\ E_1\ (a\ x)\ (b\ x)$$

where a and b are parameter nemes corresponding to E_2 and E_3. Only two cells will be allocated towards E_2 and E_3 when x is bound. After an alternative has been chosen the selected brench will be constructed and evaluated. Of course, if one or both of E_2 and E_3 does not involve x then it will not need this treatment.

Applying this optimisation to the Fibonacci example, the new code is

```
fib = Y (λfib. α fib (β fib))
α    = λfib a n. IF (< n 2) n (a n)
β    = λfib n. + (fib (- n 1)) (fib (- n 2))
```

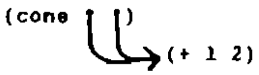
Counting the number of application cells allocated during a call of (fib 1), we find that it has decreased from 13 to 6, while the number of cells allocated per recursion for n greater than 1 has only increased from 13 to 14. This represents a great improvement in efficiency.

We have discussed this optimisation in the context of conditionals. However, it is worth replacing any large expression E that may well never be evaluated by (λx.E) x, where x is the bound variable of the enclosing λ-expression. To do this to every expression, though, would be to create combinators almost as small as Turner's ones, and so to throw away the main advantage of super-combinators. Fortunately, most expressions that the programmer writes are eventually evaluated, and so it is only in cases like the conditional that it is necessary to use this transformation.

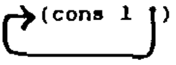
4.5. GRAPHICAL COMBINATORS

The method we have described starts from a translation of the source program into the λ-calculus and produces combinators from that. This leads to some inefficiency in the treatment of declarations: for example **let** x = 1 **in** x+2 is less efficient than 1+2, because the former is translated into (λx.x+2) 1, and thence into (κx.x+2) 1, and so requires a combinator application during its evaluation. It would be nice if these two equivalent programs were equally efficient.

We can achieve this if we extend the combinator language slightly to include declarations. In this case let x=1 **in** x+2 is a perfectly valid expression in the object language. In fact we can interpret such expressions directly as graphs. For example, let x=1+2 in cons x x can be interpreted as the graph



and letrec x = cons 1 x in x can be interpreted as

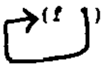


Since we are already working with graph reduction, the introduction of combinators with more complex graphs as bodies represents no real extension at all. Under this graphical interpretation of declarations the two examples we began with actually represent the same graph, and so are certainly equally efficient.

We now have combinators with general graphs as bodies, rather than trees. An example of this kind of combinator is Y, which we can define as

$$Y = \lambda f.\ \mathrm{letrec}\ x = f\ x\ \mathrm{in}\ x$$

since (Y f) is reduced to



In order to perform super-combinator abstraction on a program containing declarations, we first of all float the declarations outwards as far as possible, using the fact that

$$\lambda x.\mathrm{let}\ I=E_1\ \mathrm{in}\ E_2 \ = \ \mathrm{let}\ I=E_1\ \mathrm{in}\ \lambda x.E_2$$

provided x does not occur in E_1. This equation holds because both sides represent the same graph, and we are simply choosing the most convenient written representation of it. Having done this, we simply perform ordinary super-combinator abstraction, remembering that a variable defined in a declaration is free in a λ-expression if the expression it represents is (this requires some care in the case of **letrec** declarations). The result will be graph structured combinator code for the program

For example, starting from

```
letrec fib = λn. IF ((< n 2)
                     n
                     ((λn. + (fib (- n 1))
                             (fib (- n 2)))
                      n))
in fib
```

we derive

```
letrec fib = κn. IF ((< n 2) n (α n)
   and α = κn. + (fib (- n 1)) (fib (- n 2))
in fib
```

Both combinators are simpler and more efficient than their earlier counterparts. The improvement in efficiency is particularly great in the case of mutually recursive declarations, which under the old method were translated into the construction and subsequent destruction of a list of the values being declared.

We will not pursue this particular extension any further. As we have remarked above, the presence of combinators with graph-structured bodies does not interfere with other aspects of graph reduction, and so we can simplify the rest of this thesis by discussing tree-structured combinators only. Where relevant we will remark on the extensions necessary to cope with the more general cese.

4.8. EXPERIMENTAL RESULTS

To test the super-combinator method in practice we compared it against Turner's method. His method was chosen for the experiment because there is already considerable evidence thet it is more efficient than a lazy SECD machine [Turner79] [Peyton-Jones82]. A compiler for a high-level functional language was written, which translated into the λ-calculus and could then generate either kind of combinators. Only the methods of sections 4.2 and 4.3 were used. The code was run on a BCPL interpreter which contained precompiled definitions of Turner's combinators end could load super-combinator definitions, compiled into mechine code, if necessery. The compiler and reducer made a number of measurements, including code size, number of reductions performed during execution, total number of cells claimed, and run-time.

Ten small programs were written and benchmarked, ranging from Ackerman's function to a unification algorithm, and the results are summarised in the table below.

Program	Size	ΔCode Size	ΔReductions	ΔCells Claimed	ΔTime
1	26	36	-13	39	0
2	36	-10	-48	46	0
3	49	0	-42	10	-12
4	51	-5	-47	32	0
5	75	9	-36	9	-3
6	93	-9	-42	-16	0
7	106	-9	-23	-19	-21
8	115	-13	-30	-8	-11
9	307	-7	-39	21	-17
10	317	-31	-60	-35	-45

The figures in the table are the percentage change in moving to super-combinators.

These results do not demonstrate an awesome superiority. However, it should be borne in mind that all the example were very small, and that the advanteges of super-combinators should become more pronounced for larger programs (since small programs tend to compile to small super-combinators, the advantage of "large execution steps" is lost). The table is arranged in order of increasing program size, and there is a visible improvement as we look down the columns. The anomalies in the "cells claimed" column are probably due to the fact that we did not use the technique of section 4.4, which affects particularly programs 2, 4 and 9. We are heartened that no program ran more slowly when compiled to super-combinators. Therefore we are reasonably confident that the use of super-combinators will produce a significant improvement in the performance of real programs. For details of the experiments, see the Appendices.

CHAPTER 5

SUPER-COMBINATOR COMPILERS

5.1. INTRODUCTION

In chapter 4 we described how functional programs can be translated into super-combinators, but we did not give an algorithm for this translation. We have not yet demonstrated that it can be done reasonebly efficiently. In section 5.2 we give an informel description of such an algorithm, which converts a λ-expression into super-combinators in a single pass, and in the following three eections we outline three different implementationsof this algorithm in imperative, functional and logic languages. Finally, in section 5.7 we will review our experience of implementing a reasonably complex algorithm in the three different styles.

5.2. THE ALGORITHM

We first describe an algorithm that incorporates the techniques of sections 4.2 and 4.3. The optimisation of section 4.4 is easily added, and the generation of graph-structured combinators has already been discussed.

Since the elgorithm is going to order combinator parameters optimally, it will need to work with the netive λ-expressions of parts of the program. We begin by observing that the native λ-expression of any expression can be identified by a single number. This is because the λ-expressions enclosing a particular program point form a sequence, and so we may refer to them by their position in this sequence. Thus, 1 refers to the outermost λ-expression, 2 refers to the next one in *etc.* We may therefore associate a number identifying its native λ-expression with every expression in the program. We call this number an expression's *lexical level*, or sometimes just its *level*. It is convenient to assign constant expressions a level of zero, since this corresponds to a notional λ-expression enclosing the whole program.

We can compute the lexical level of every expression in the program in a single recursive pass, as follows. We assign constant expressions a level of zero, and we deal with identifiers using an environment which maps them to their lexical level (the number corresponding to the λ-expression binding them).

We assign applications the maximum of the lexical levels of the function and argument. This is justified because the maximum corresponds to the innermost of the two native λ-expressions. It is clear that the application is not free in this λ-expression, but is free in all inner λ-expressions, and so this rule correctly identifies the application's native λ-expression.

Determining the lexical level of a λ-expression is more complex. We notice, though, that the problem can be avoided if we can replace λ-expressions by corresponding applicative forms "on the fly". In this case we can scan the body of a λ-expression, replace it by the corresponding super-combinator application, and then compute the lexical level of this application using the rules above. This is the approach we will take.

We may summarise our conclusions so far as follows. The program syntax tree is scanned in depth-first order. As soon as any sub-tree has been scanned, its lexical level is determined, and it is converted into the equivalent combinator form. When the scan terminates, the whole program will have been compiled into super-combinators.

Before explaining how λ-expressions are compiled on the fly, we consider the problem of identifying maximal free expressions. We are interested, not just in identifying the mfes of the nearest enclosing λ-expression, but in identifying all the expressions in the program which are mfes of *any* λ-expression. Fortunately this can be done using only the level numbers.

For an expression to be maximal free it must first be free in some λ-expression, *ie* its level number must be less than that of the nearest enclosing λ-expression, and secondly it must be maximal. This means that its level number must differ from the level number of the immediately enclosing expression (or it must be the entirety of the body of a λ-expression). Otherwise, both it and the immediately enclosing expression would be free in all the same λ-expressions and it could not be maximal free in any. We can even identify the λ-expression it will be maximal free in; it has to be one in which it is free, but in which the enclosing expression is not free. Therefore it is maximal free in the native λ-expression of the enclosing expression.

We may summarise these rules by: an expression is an mfe if it is the body of a λ-expression and its lexical level is less than the level of the λ-expression, or if it is a function or argument and its lexical level is less than the level of the application it forms part of. The expression is an mfe of the λ-expression it is the body of in the first case, and the native λ-expression of the application in the second case.

Therefore, once we have scanned the body of a λ-expression we will have identified all its maximal free expressions. This tells us what the parameters of the corresponding super-combinator will be. We need to sort them into the optimal order, but this is easy because the order depends on which native λ-expressions enclose which others, and this can be determined by comparing level numbers. We can order the parameters optimally by sorting them into order of increasing lexical level.

Now we can replace the λ-expression by a super-combinator applied to the mfes, and construct the combinator by replacing the mfes in the λ-expression's body by appropriate parameter names (numbers in real implementations). This completes the description of our algorithm.

For the most part, this algorithm is fairly simple. However, it is quite tricky to replace mfes by argument names, as we gaily said in the last paragraph, in an efficient way. The next three sections outline implementations of this algorithm in imperative, functional and logical styles, and differ primarily in the solutions adopted to this problem.

5.3. AN IMPERATIVE COMPILER

We shall describe our imperative compiler by giving a Pascal-like skeleton and leaving the reader to fill in the details. We begin by giving the type of tha syntax tree nodes:

```
type node = record level: integer;
            case kind of
            constant:    (...);
            variable:    (...);
            application: (fn, arg: node);
            lambda:      (bvar, body: node);
            argument:    (argnum: integer);
            combinator:  (numargs: integer;
                          body: node)
            end;
```

Since the compiler operates by physically transforming the original syntax tree into the combinator version there is provision for storing a level number in the node and there are alternatives to represent combinators.

Notice that we refer to combinator arguments by integers. These correspond to stack offsets during execution, and their precalculation means that combinator arguments do not have to be looked up in an environment at run-time. This also means that we can represent a super-combinator by its body and a count of its arguments.

We write node construction functions in upper case by convention, and we use three of them: APPLY to construct an application node, SUPER to construct a combinator node, and ARG to construct a combinator argument name.

The compiler maintains ~~two~~ three important global variables, whose declarations are:

```
        var cl: integer := 0;
            mfes: array [1..] of
                         sequence of address of node;
            bvs: array [1..] of sequence of address of node;
```

cl is the current level, initially zero. It is incremented when the compiler begins to scan s λ-expression snd decremented afterwards, so it always holds tha nesting depth of the nearest enclosing λ-expression. mfes is used to atore maximsl frea axprassions as they are found. It is an array with one element per λ-expression enclosing the current node. The element holds a sequance of all the mfes of that λ-expression found so far. In fact, it is the addresses of the mfes which are held in the sequence; this permits the replacement of mfes by argument names alluded to above. bvs holds the addresses of all occurrences of bound variables, so that they can be replaced by numbered arguments.

The compiler itself is the following procedure:

```
procedure compile (var n:node; e: env);
  if isconstant (n) then n.level := 0
  elif isvariable (n) then
    n.level := lookup (n, e); augment(bvs[n.level],address n)
  elif isapplication (n) then
    compile (n.fn, e); compile (n.arg, e);
    n.level := max (n.fn.level, n.arg.level);
    for cpt in {fn, arg} do
      if n.cpt.level < n.level then
        augment (mfes [n.level],
                  address n.cpt)
      fi
    od
  elif islambda (n) then
    cl := cl + 1; mfes [cl] := <>; bvs[cl] := <>;
    compile (n.body, bind (n.bvar, cl, e));
    sortmfes (mfes [cl]);
                                      +1
    n := SUPER (length (mfes [cl]) ^, n.body);
    for i in 1..length (mfes [cl]) do
      n := APPLY (n, deref mfes [cl] [i]);
      n.level := n.arg.level;
      if n.fn.level < n.level then
        augment (mfes [n.level], address n.fn)
      fi;
      deref mfes [cl] [i] := ARG (i)
    od;                  | for i in 1..length(bvs[cl]) do
    cl <-:= cl - 1 ------|    deref bvs[cl][i] :=
  fi                     |       ARG(length(mfes[cl])+1)
end.                     | od
```

Notes: n is a var parameter so that the compiler can overwrite it with the compiled version of the node. The environment is manipulated with bind, which returns an environment equal to the given one except that the given variable is mapped to the given integer, and lookup which returns the level of a variable. augment adds an element to the end of a sequence. address returns the address of a variable, and deref permits that variable to be read or updated via its address. sortmfes sorts a sequence of mfe addresses into optimal order, according to the criterion explained above.

The essential feature of this compiler is the way that it stores mfe addresses so that it can both fetch the mfe to construct the replacement applicative form, and update the reference to it in the function body to refer to an argument name instead.

5.4. A FUNCTIONAL COMPILER

As in the last section, we shall begin by describing the types we use. There are several types in the functional compiler, since we cannot use assignment to store all the information in the same node. We use a syntax like that of HOPE [Burstall80] to describe them.

The input and output from the compiler are NODEs:

```
        data NODE = VAR(...) |
                    CONST(...) |
                    APPLY(NODE,NODE) |
                    LAMBDA(NODE,NODE) |
                    ARG(NUMBER) |
                    SUPER(NUMBER,NODE)
```

The recursive compile function itself must have several results. since we expect it to return the level of an expression, find its mfes, and return a version of the expression with all mfes replaced by ARG(n) nodes. We encapsulate them in the type EXPR:

```
data EXPR = EXPR(NUMBER,EXPR-LIST,NODE)
```

Notice that it is the compiled form of the mfes that is returned.

Since we expect the compiler to replace mfes by ARG(n) nodes, and since the names to be used depend on the context in which the expression being compiled occurs, we must pass the compiler an argument giving the names to use for each mfe found. We use the NAME type to pass the name of an mfe, and the names of all its sub-mfes together.

```
data NAME = NAME(NUMBER,NAME-LIST)
```

Finally, environments have type ENV which is NODE→NUMBER£NUMBER. Given a VAR-NODE they return a level number and an argument number to be used as a replacement.

The type of the compiler is

```
compile: ENV→NAME-LIST→NODE→EXPR
```

Since the NAME-LIST gives the names of the mfes found, it will always have an isomorphic structure to the EXPR-LIST part of the result. This isomorphism will extend to all levels of the structures.

We shall describe the compile function in a SASL-like notation [Turner76], with a few extensions. Its skeleton is:

```
letrec compile env namee exp =
   case exp of

   VAR(...) → ...
   CONST(...) → ...
   APPLY(f,a) → ...
   LAMBDA(v,e) → ...
   esac
```

Compiling variables and constants is easy: the relevant parts of compile are

```
VAR(...) → let lev, num = env exp in
                EXPR(lev, [], ARG(num))
CONST(...) → EXPR(0, [], exp)
```

Applications are harder to deal with since the compiler must decide whether either the function or the argument is an mfe, and if so replace it by its name. Getting the NAME-LIST parameter right to the recursive calls of compile is also tricky. We shall first show the necessary program and then explain it.

```
APPLY(f,a) →
  letrec EXPR(flev,fmfes,fnew) =
       compile env fnames f
     and EXPR(alev,amfes,anew) =
       compile env anames a
     and result, fnames, anames =
       flev=alev →
          (EXPR(flev,fmfes++amfes,
                APPLY(fnew,anew)),
           take(£fmfes)names,
           drop(£amfes)names);
       let NAME(num,subnames):names' = names
       in
          flev<alev →
             (EXPR(alev,
                 EXPR(flev,fmfes,fnew):amfes,
                 APPLY(ARG(num),anew)),
              subnames, names');
          flev>alev →
             (EXPR(flev,
                 EXPR(alev,amfes,anew):fmfes,
                 APPLY(fnew,ARG(num)),
              names', subnames)
  in result
```

Notes: The unusual functions used in this expression are as follows: ++ is append. : is cons. £ is length of a list, and take and drop return the first n elements and all but the first n elements of a list respectively.

Notice that compile computes fnames and anames from the results of the recursive calls, even though they are themselves parameters to those calls. It guarantees that if the original call had a names parameter isomorphic to its EXPR-LIST result, then the recursive calls will have too. This circular style of programming is only possible because of lazy evaluation, and even so, one must convince oneself that the function is actually defined. It is easy for functions written in this way to fail to terminate. In the case of compile, one can argue that all level numbers are obviously defined, and so all mfes are too. Finally, this implies that all names and new expressions are also defined and so compile always terminates.

λ-expressions are compiled by the expression below.

```
LAMBDA(v,e) →
   letrec vlev, env' = bind v vnum env
      and emfes, enew =
         letrec EXPR(elev',emfes',enew') =
                   compile env' enames' e
            and enames' =
               vlev=elev'→enames; subnames
            and [NAME(num,subnames)] = enames
         in vlev=elev' →
               emfes', enew';
               [EXPR(elev',emfes',enew')],
                   ARG(num)
      and orderedmfes, permutation =
         let s =
            sort (λ(?,EXPR(alev,?,?))
                      λ(?,EXPR(blev,?,?)).
                         alev>blev)
                 (zip [1..£emfes] emfes)
         in map (λ(i,e).e) s, map (λ(i,e).i) s
      and result, orderednames, vnum =
         mkap comb orderedmfes names
      and comb = EXPR(0, [], SUPER(vnum,enew))
      and enames =
         map (λ(i,n).n)
             (sort (λ(i,?)λ(j,?). i<j)
                   (zip permutation
                        orderednames))
   in result
```

First, we add the bound variable to the environment using bind, which we assume also assigns and returns the new level number. Then we compile the body of the λ-*expression* and take account of the fact that the whole body might be an mfe (if vlev is greater than elev'). Notice that the declaration of num and subnames can only be executed when enames has precisely one element. This is perfectly acceptable, since in other cases neither num or subnames is used and so the declaration does not need to be executed.

We go on to sort the mfes found into optimal order, and record the permutation used so that the corresponding *names* can be restored to the original order later. The question mark used as a variable name means that the variable is not actually required. sort takes a partial order and a list and sorts the list into increasing order. zip takes two lists and returns the corresponding list of pairs.

We can then use the subsidiary function mkap, which takes a combinator and a list of arguments (in reverse order) and constructs the application of the combinator to the arguments. It also needs to be passed the names corresponding to the mfes of the result, of course, and it returns a list of names to be used to replace the arguments and the next available argument number.

Finally, we can construct the super-combinator itself and sort the names to replace the arguments into the original argument order.

It only remains to define mkap. We do so below with no further explanation, since it is very similar to the APPLY case of compile.

```
letrec mkap f [] [] = (f, [], 1)
   and mkap f
           (EXPR(alev,amfes,anew):args)
           rnames =
      letrec EXPR(flev,fmfes,fnew),
              anames',
              nextarg =
             mkap f args fnames
         and NAME(num,subnames):rnames' =
             rnames
         and r, fnames, anames =
             flev=alev →
                (EXPR(alev,
                      amfes++fmfes,
                      APPLY(fnew,anew)),
                 drop (£amfes) rnames,
                 take (£amfes) rnames);
             flev<alev →
                (EXPR(alev,
                      EXPR(flev,fmfes,fnew):amfes,
                      APPLY(ARG(num),anew)),
                 subnames,
                 rnames')
      in (r,
          NAME(nextarg,anames):anames',
          nextarg+1)
```

5.5. A LOGIC COMPILER

We present our logic programming compiler in Prolog [Kowalski79]. The datatypes used correspond to the ones in the functional version, so we shall not describe them further. The compile predicate takes the form

```
compile(env,node,expr,names)
```

where env is an environment, node is the original expression, expr is a structure with functor EXPR containing the level, mfes and new form of the expression, and *names* is a list of names isomorphic to the *mfe* component of expr.

compile is *defined* by four clauses, the first two of which compile variables and constants. They are:

```
compile(env,VAR(...),EXPR(l,[],ARG(n)),[]) :-
    lookup(env,VAR(...),l,n).
compile(_,CONST(...),EXPR(0,[],CONST(...)),[]).
```

We have assumed that lookup finds the level and corresponding argument number of a variable from the environment, and that the underline is an anonymous variable.

Applications are compiled by the clause

```
compile(env,APPLY(f,a),expr,names) :-
    compile(env,f,fexpr,fnames),
    compile(env,a,aexpr,anames),
    apply(fexpr,aexpr,fnames,anames,expr,names).
```

The function and argument are compiled first, and then apply is used to combine the results. If the function and argument have the same level, then apply simply applies one to the other

```
        apply(EXPR(l,fmfes,fnew),EXPR(l,amfes,anew),
                fnames,anames,
                EXPR(l,mfes,APPLY(fnew,anew)),names) :-
            append(fmfes,amfes,mfes),
            append(fnames,anames,names).
```

Otherwise the one with the lowest level is made into an mfe:

```
        apply(EXPR(fl,fmfes,fnew),EXPR(al,amfes,anew),
              fnames,anames,
              EXPR(al,EXPR(fl,fmfes,fnew).amfes,
                   APPLY(ARG(n),anew)),
              NAME(n,fnames).anames) :- fl<al.
        apply(EXPR(fl,fmfes,fnew),EXPR(al,amfes,anew),
              fnames,anames,
              EXPR(fl,EXPR(al,amfes,anew).fmfes,
                   APPLY(fnew,ARG(n))),
              NAME(n,anames).fnames) :- fl>al.
```

λ-expressions are compiled by the clause:

```
        compile(env,LAMBDA(v,e),expr,names) :-
            bind(env,v,nextarg,env',lev),
            compile(env',e,eexp',enames'),
            lambody(lev,eexp',enames',
                    EXPR(elev,emfes,enew),enames),
            sortmfes(emfes,enames,sortedmfes,sortednames),

            mkap(EXPR(0,[],SUPER(nextarg,enew)),
                sortedmfes,sortednames,
                expr,names,nextarg).
```

First we bind v into the environment, giving it argument number nextarg. We assume that bind computes both the new environment env' and the new level number lev, as in the functional version. Then we compile the body, and take account of the fact that it might form an mfe by itself (in lambody). sortmfes sorts the mfes and names into optimal order, and finally mkap is used to construct the replacement expression, assign names and compute the number of arguments of the combinator.

lambody is defined by

```
        lambody(lev,EXPR(lev,mfes,new),names,
                    EXPR(lev,mfes,new),names).
        lambody(lev,EXPR(l,mfes,new),names,
                    EXPR(lev,[EXPR(l,mfes,new)],ARG(n)),
                    [NAME(n,names)]) :- l<lev.
```

which just makes the body of the λ-expression into an mfe if it doesn't contain the bound variable.

mkap applies the combinator to its arguments and assigns their names. Its form is

```
        mkap(comb,mfes,names,expr,enames,nextarg)
```

where comb is the combinator to be applied, mfes is its arguments, names the names to be assigned to those arguments, expr the resulting expression, enames the names of mfes of that expression, and nextarg the next free argument number. It is defined by the clauses:

```
        mkap(f,[],[],f,[],1).
        mkap(f,arg.mfes,NAME(nextarg',anames).names,
                e,enames,nextarg) :-
            mkap(f,mfes,names,e',enames',nextarg'),
            apply(e',arg,enames',anames,e,enames),
            nextarg is nextarg'+1.
```

We have made great use of the Prolog variable in this program to distribute the computation of name structures and argument numbers to convenient places. We feel that this program demonstrates the power of Prolog rather well: such things as sorting mfes and names proved considerably simpler than in the functional equivalent. We have also used this style to include an optimisation in the compiler, which removes repeated mfes from the parameter list. This can be done by the predicate

```
        optimise(mfes,names,mfes',names')
```

which takes an unoptimised mfe list in mfes and names, and computes the optimised equivalent in mfes' and names'. It is defined as

```
optimise([],[],[],[]).
optimise(mfe.mfes,name.names,
        mfe.mfes',name.names') :-
    not(member(mfe,mfes)),
    optimise(mfes,names,mfes',names').
optimise(mfe.mfes,name.names,mfes',names') :-
    element(mfes,i,mfe), element(names,i,name),
    optimise(mfes,names,mfes',names').
```

where element(list,el,index) is true if el is the indexth element of list. The corresponding optimisation in the functional compiler was too complicated to include.

5.8. CONCLUSION

The super-combinator abstraction algorithm described in this chapter is reasonably complex, and so it is interesting to compare our three different implementations. Curiously, the imperative program is the shortest and requires least explanation. This is partly because we assumed that many facilities of functional languages were available, which is not usually the case. However, the imperative solution represents all its information in one complex data-structure, rather than breaking it down into simpler units, and it depends crucially on side-effects happening at the right times. We contend that these make it difficult to understand, even though it is relatively short. Of course, they also render it totally unsuitable for a parallel implementation.

The functional and logic programs are very similar. However, the isomorphism between name structures and mfe structures (which is crucial to both of them) is concealed in the functional version, since the name structures appear as arguments and tha mfe structures appear as results. This renders the structure of the functional version more obscure. We consider this strong evidence that the need to specify direction of information flow can occasionally lead to badly structured programs. Except for this point we found the functional style more expressive than Prolog. Both the functional and logic versions are divide-and-conquer programs, and so both are suitable for implementation on a parallel machine.

We would have liked to claim that the experience of implementing the same complex algorithm in three differant languages demonstrated conclusively the superiority of functional and logic programming over imperative programming. Unfortunately this example is not terribly conclusive. The Pascal hacker may well claim that the imperative implementation is easier to understand because it is shorter, but this is because he is used to thinking in terms of side-effects and time. However, our own understanding of the abstraction algorithm was advanced considerably by the experience of writing the functional and logic versions, while writing the imperative progrem served only to confuse us. In this respect we feel that the axample has demonstrated the superiority of declarative programming, if only to ourselves.

CHAPTER 6

ANALYSIS OF EFFICIENCY

6.1. INTRODUCTION

In this chapter we attempt a theoratical analysis of the efficiency of supar-combinators. Although it is difficult to obtain concrete results in this area, we have two results which we think significant.

We considar translation of a program of size n into super-combinators and into Turner's combinators, and we find the order of the size of code produced. Burton has already shown that the combinator code need be no larger than O(nlogn) [Burton82]; we will show thet, on average, it is indeed this large. We will also show that the super-combinator code can be no larger than O(nlogn), but that it may sometimes be this large, and we will offer some evidence that it is usually smaller.

This problem is interesting for two reasons. Firstly, in early implementetions using Turner's combinators, excessive code size was a serious problem. Some small programs compiled into code so large that it could not possibly be run. We would like to demonstrate that this cannot happen using super-combinators. Secondly, at least in "straight line" programs, code size is an approximate measure of execution speed. We use this to make a (very vague) comparison with the SECD machine.

6.2. TURNER'S COMBINATORS

We begin by quoting the appropriate results for Turner's combinators. Kennaway has shown that the translation method Turner describes can lead to the production of code of size $O(n^2)$ [Kennaway82]. Burton has given an improved method and shown that it produces code of size $O(n\log n)$ in the worst case [Burton82]. It is easy to see that this result is optimal, and we give a proof here.

We take the size of an expression to be the number of nodes in its syntax tree, since this is consistent with the graph reduction applications we have in mind. This is equivalent to the number of symbols in its written representation, not counting brackets. In this definition we differ from Burton, who counts the lengths of identifiers in his size measure. (However, we have translated his result, given above, into our terms).

Now, since Turner uses a fixed set of combinators, S different ones say, it is clear that there are at most S^n different combinator expressions of size n. However, since a λ-expression may contain O(n) different symbols, there may be $O(n^n)$ different λ-expressions of size n. In fact, we can exhibit n^n non-interconvertible λ-expressions of size O(n), being

$$\lambda v_1 \ldots v_n.\ \mathrm{CONS}\ w_1\ (\ldots\ (\mathrm{CONS}\ w_n\ \mathrm{NIL})\ \ldots)$$

where all the v_i are different identifiers, and each w_i is one of the v_j. These λ-expresslons are clearly non-interconvertible since they all yield different results when applied to n different arguments.

Now, each of the n^n λ-expressions we have given must compile to a different combinator expression, so if N is the worst case code size then there must be at least n^n different combinator expressions of size no more than N. We must therefore have

$$n^n < S^N$$

and so, taking logarithms,

$$n\log n < N\log S$$

That is, N is at least O(nlogn). We can actually derive a stronger result from this argument: no matter how good the compiler, almost all λ-expressions must compile to code of size at least O(nlogn). Burton's result therefore tells us that the *average* code size of a λ-expression after translation to combinators is O(nlogn).

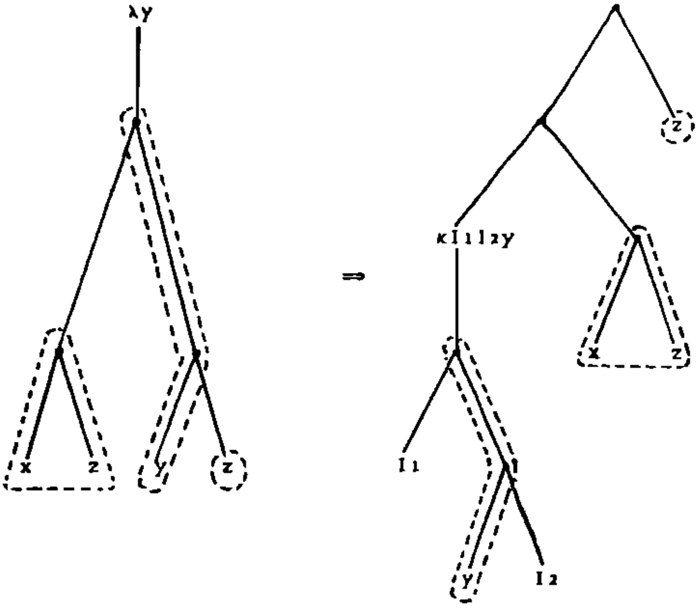
6.3. SUPER-COMBINATORS

An analysis of the complexity of the code size of super-combinators is much more difficult. We will begin by observing that translation to super-combinators does increase the size of the program, and by identifying the source of the increase.

Consider a single λ-expression λx.E, and suppose it has mfes E_1 to E_n. It will be translated into

$$(\kappa I_1 \ldots I_n x.\ E[I_i/E_i])\ E_1 \ldots E_n$$

If we consider the syntax tree of the body of the original λ-expression, we see that it is broken up into the mfes, and the non-mfe part. The parts all reappear in the code, the mfes as super-combinator arguments, and the non-mfe part as the super-combinator body. The diagram below illustrates this, using dotted lines to delimit the parts of the tree.



Since super-combinator arguments are actually represented by integers it is only necessary to store the arity of a super-combinator at its head, not the names of its arguments. $\kappa I_1 \ldots I_n x.E$ would therefore be represented as $\kappa n.E$, and so we count the κ and argument names as one node, the same as λx.

It appears that the *only* nodes in the code which do not correspond directly to nodes in the original source are those used to apply the combinator to its arguments. There is one of these for each *mfe*, and so we may conclude that when a program is translated into super-combinators its size increases by the total number of *mfes* found during translation.

Since accounting for *mfes* is so central to our problem it is worth studying them further. We have already pointed out that an *mfe* is an *mfe* of the native λ-expression of its immediately enclosing expression. To see why, suppose E_0 is an *mfe*, and its immediately enclosing expression is E_1. Clearly E_0 and E_1 have different native λ-expressions, and the native λ-expression of E_0 encloses that of E_1. Since E_1 is free in all λ-expressions enclosed by its native one, E_0 cannot be maximal free in any of them. However, since E_1 is not free in its native λ-expression, and E_0 is, then E_0 is maximal free in it. In implementation terms this corresponds to observing that, since E_1 will be passed *from* its native λ-expression to its proper location, E_0 will be carried along inside it and need not be passed separately.

Moreover, each *mfe* will only be an *mfe* of one λ-expression. That *is*, it will not be an *mfe* of the expression produced when the λ-expression it is an *mfe* of is compiled. This is because, when that λ-expression is translated into a combinator application, the combinator parameters are ordered by the optimal ordering given in section 4.3. Consider E_i in

$$\alpha \; E_1 \; \ldots \; E_i \; \ldots \; E_n$$

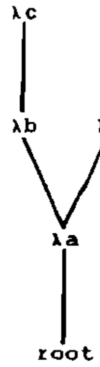
By the optimality criterion, all the E_j to the left of E_i have native λ-expressions enclosing or equal to the native λ-expression of E_i, and so $(\alpha \; E_1 \ldots E_i)$ has the same native λ-expression as E_i. This means that E_i is not an *mfe* of any other λ-expression, although $(\alpha \; E_1 \ldots E_i)$ may be.

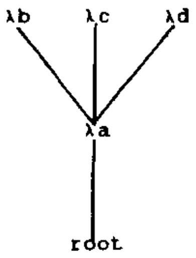
So, now we know that we need only count mfes, each of which will only occur once. However, the situation is complicated by the fact that expressions introduced during translation may themselves turn out to be mfes of further λ-expressions. For example, if a λ-expression has two mfes E_1 and E_2 with different native λ-expressions, then it will be replaced by $(\alpha\ E_1\ E_2)$, and $(\alpha\ E_1)$ will itself be an mfe of the native λ-expression of E_2. Not only this, $(\alpha\ E_1\ E_2)$ may be an mfe of the next enclosing λ-expression. Either of these new mfes may then cause enclosing λ-expressions to have still further new mfes. The problem of analysing the code size of super-combinators is accounting for these mfes generated during translation, as well as those originally present in the program.

6.4. ACCOUNTING TREES

In order to keep track of the generated mfes we introduce the notion of an *accounting tree.* An accounting tree for a program is a tree whose vertices are the λ-expressions of the program, and whose edges are constrained to connect a λ-expression to an enclosing one. By convention we draw accounting trees growing upwards with the notional outermost λ-expression at the bottom, so, for example, the diagram below shows two possible accounting trees for

$$\lambda a.(\lambda b.\lambda c.b)(\lambda d.a)$$





In our diagram, edges connect λ-expressions above to enclosing λ-expressions below.

For any program, there is a "tallest" accounting tree which we call the *initial* accounting tree, in which all λ-expressions are connected to their immediately enclosing λ-expressions. The tree on the left above is the initial accounting tree for the given program.

Starting from a program's initial accounting tree, we will use the mfes originally present in the program, one by one, to transform the accounting tree so that we can discover from it which further mfes will be generated as a consequence of the original ones. The number of additional mfes due to an original one will also be a reasonable measure of the "cost" of the associated transformation to the tree. This will allow us to translate the problem of super-combinator code size into an equivalent one concerning the cost of a tree-transformation algorithm.

During the transformation process we will preserve the following invariant: each λ-expression will be connected to the native λ-expression of its most recently discovered mfe, original or generated, unless none of its mfes have been discovered yet, in which case it will be connected to its immediately enclosing λ-expression. The order in which original mfes are used to transform the tree will be constrained by the following rule: if E_1 and E_2 are mfes such that the native λ-expression of E_2 encloses (is nearer the root of the tree than) the native λ-expression of E_1, then we will use E_1 before we use E_2. This constraint means that we will always discover the mfes of any particular λ-expression in reverse order, although not necessarily one after another. This property is crucial to the proof.

Now, consider processing an original mfe E of λ-expression A. Let N be E's native λ-expression. We must at least remove the edge leading downwards from A and connect A to N instead, in order to preserve our invariant. Suppose B is the λ-expression immediately below A before this modification. Then we know that A will be replaced by a combinator expression

$$\alpha \; \ldots \; E \; E' \; \ldots$$

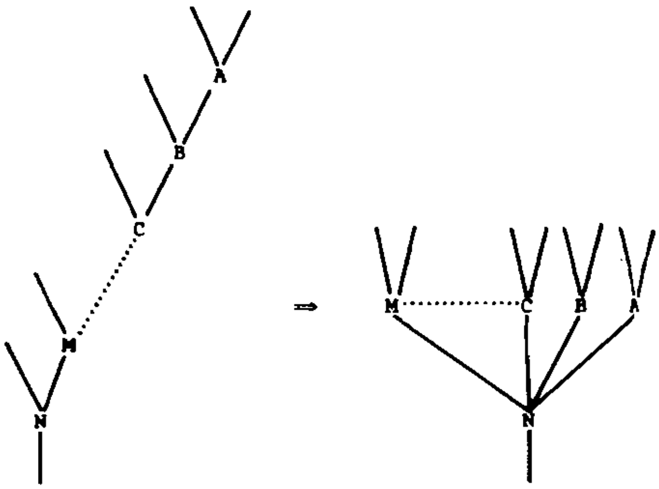
where E' has native λ-expression B, or that A will be replaced by

$$\alpha \; \ldots \; E$$

and B is A's immediately enclosing λ-expression. We know this because of the optimal ordering of mfes, and because mfes are discovered in reverse order as noted above. In either case,

$(\alpha \; . \, . \, . \; E)$

will be a generated mfe of B. So, B must also be reconnected to N, and the same argument applies to C, the λ-expression immediately below B, and so on. There must be a chain of λ-expressions B,C,D,... leading downwards from A through the accounting tree, each of which should be reconnected directly to N, and each of which has a generated mfe with native λ-expression N. This chain must terminate somewhere, and cannot terminate at the root of the accounting tree since the root notional λ-expression cannot possibly have any mfes at all. It must terminate instead at N. Strictly speaking, it terminates at a λ-expression just above N, M say, since M already has an mfe with native λ-expression N and so no new mfe is actually generated. The transformation that results is illustrated in the diagram below.



So, if the path length from A to N is n, then the presence of an original mfe of A with native λ-expression N implies the presence of n−3 generated mfes. We associate a "cost" of n with performing the associated transformation of the accounting tree, so the total increase in size of a program when translated to super-combinators will be lese than the total cost of the transformations applied to its accounting tree.

We may therefore conclude that when a program is translated into super-combinators the increase in its size is less than the maximum total cost of applying as many "flatten" operations as the program originally contains mfes to a tree of as many nodes as the program contains λ-expressions, where a flatten operation consists of selecting a path in the tree leading towards the root and reconnecting every other node on the path directly to the end nearest the root, and where the cost of a flatten operation is the length of the path.

6.5. UNION-FIND ALGORITHMS

We break off from analysing super-combinators now to describe the union-find problem and algorithms for its solution. It will turn out that previous analyses of these algorithms will enable us to complete our analysis of super-combinator code size.

The UNION-FIND problem concerns the manipulation of a number of disjoint sets which partition a universe. Initially the universe is partitioned into singletons, and thereafter two kinds of operation may be applied: UNION(A,B,C) which unites sets A and B into a new set called C, and FIND(x) which determines which set the element x belongs to.

The basic UNION-FIND algorithm represents the sets as trees of elements with set names at the roots. UNIONs are performed by making the root of the tree representing one set point at the root of the tree representing the other. FINDs are performed by following pointers until one reaches a root. UNION is therefore of constant cost, and FIND of cost proportional to the length of the path followed. Two optimisations can be applied to this algorithm:

The Collapsing Rule: when a FIND is performed, all nodes on the path to the root are made to point directly at the root, speeding up subsequent FINDs.

The Weighting Rule: a UNION operation always makes the set containing fewer elements point at the set containing more. This will also tend to speed up subsequent FINDs.

It turns out that, for the purposes of analysing these algorithms, we can consider all UNIONs to occur first, building a tree, followed by *partial* FINDs, which start at an element and follow a path part of the way to the root. It should be clear that the algorithm using only the collapsing rule performs exactly the same kind of transformations as our manipulations of accounting trees in the last section, and moreover the cost measures are the same in both cases.

Tarjan has analysed these algorithms [Tarjan75] and his results are that, if $m > n$ partial FINDs are performed on a tree of n nodes then the total cost, $t(m,n)$, will satisfy

$$t(m,n) = O(m.\max(1,\log(n^2/m)/\log(2m/n)))$$

If only the collapsing rule is used, and

$$t(m,n) = O(m.\alpha(m,n))$$

if the collapsing and weighting rules are both used, where α is related to the inverse of Ackermen's function and grows so slowly that it is bounded by a small constant for all practical purposes.

Fischer has given an example [Fischer72] where

$$t(n,n) \propto n\log n$$

using only the collapsing rule, showing the former upper bound tight in the case m=n.

6.6. CONCLUSION

We may therefore conclude that the increase in the size of a program containing n λ-expressions and m>n original mfes is bounded by

$$O(m.\max(1,\log(n^2/m)/\log(2m/n)))$$

Letting $\alpha=m/n$ be the average number of original mfes per λ-expression, this bound can be reexpressed as

$$O(m.\max(1,\log(n/\alpha)/\log 2\alpha))$$

It is clear that, for $\alpha>1$, $\log(n/\alpha)/\log 2\alpha$ decreases as α increases, and it is equal to 1 for $\alpha=\sqrt{(n/2)}$. Therefore, for $\alpha>\sqrt{(n/2)}$ the bound is linear in m, and therefore linear in the size of the program. For $1 < \alpha < \sqrt{(n/2)}$, the bound is less than the value at $\alpha=1$, $O(n\log n)$. In fact, $\log(n/\alpha)/\log 2\alpha < k$ when $\alpha > ({}_{k+1}\sqrt{(2n)})/2$, so for these values of α the bound is km. For $\alpha < 1$, we observe that decreasing the number of FINDs cannot possibly increase the overall cost, which is therefora still bounded by $O(n\log n)$.

Therefore the code size of a program of size N when translated to super-combinators is bounded by $O(N\log N)$. Fischar's example shows that there are programs whose code is this large; however, his axample is highly symmetric and is not a likely structure for a real program. For programs where the average number of mfes per λ-axpression is large enough then the code size is linear in the program size. For programs "balanced" in the sense that their initial accounting trees could be constructed by a sequenca of UNIONs satisfying the weighting rule a much tighter, almost linear, bound applies Lastly, these are worst-case bounds, and we have been told that in practice even the collapsing rule is sufficient to make a UNION-FIND algorithm run in nearly-linear time. This suggests that the average behaviour is better than $O(n\log n)$, and that in practice super-combinator code size will be nearly linear in the size of the program.

We believe that the code size of combinator and super-combinator implementations reflects a "slow-down factor", whereby the same expression may take longer to execute if it is part of a large program than if it is part of a small one. We believe code size is a good measure of this slow-down factor because, in each kind of implementation, the time to apply a combinator is proportional to its size; there ara no "instructions" which may take a variable amount of time to execute. It might appear that this slow-down is present only in these implementations, but in fact it is also prasent in the SECD machine, and in conventional implementations of languages like Pascal. In these cases it appears as the time for an environment lookup, which varies with the number of names in the environment. Because there is no direct way of relating tha average number of names in scope with the size of a program it is difficult to estimate the slow-down factor of the SECD mechine, but we observe that it is possible to write programs of size n with a slowdown factor of O(n) for Henderson's SECD machine [Henderson80], and it seems reasonable to us that the average number of names in scope should increase at least with the logarithm of the program size, giving a slow-down factor of at least O(logn). Since the maximum possible slow-down factor for a super-combinator implementation is O(logn), this suggests super-combinators may be an inherently more efficient implementation method than the SECD machine.

We conjecture that any implementation method for the λ-calculus (or any other language with Algol-like scope rules) must have a slow-down factor, and that in tha worst case this factor will be at least O(logn).

CHAPTER 7

EVALUATION ORDER

7.1. INTRODUCTION

In chapter 2 we argued that one of the most important advantages of functional programming languages is that they relieve the programmer of the burden of expressing a desired evaluation order through the structure of his program. In this chapter we exhibit cases where a particular evaluation order is critical to the efficiency of the program, and we suggest structure-independent ways of controlling it.

Let us first consider how control over evaluation order is used to improve efficiency in imperative languages. We take as an example a program that reads a file and prints the number of capital As in it. One possible evaluation order is to read the whole file into memory first, and then count all the As in it and print the result. This would be programmed as:

```
read the file;
count the As
```

Alternatively, we could read the file one character at a time and update the count as we go. This would be programmed as:

```
open the file;
zero the count;
until end of file do
    read a character;
    if it's an A then increment the count fi
od;
close the file;
print the count
```

This latter program is much less modular than the former: the two logically independent operations of reading the file and counting the As have had to be programmed together. However, it is much more efficient because it needs only a constant amount of space to run in. The former method may requires an amount of space which depends on the size of the file. This is such a gross difference that we cannot afford to ignore it.

So evaluating a program in the correct order can be critical to space efficiency. Since we cannot (and do not want to) express an order through the structure of a functional program, we must ask ourselves: firstly, is lazy evaluation (or any other strategy already in use) the correct choice anyway, and if not, how can we best cause the correct order to be used? In section 7.2 we will exhibit examples which demonstrate that lazy evaluation is sometimes a poor choice. In section 7.3 we will give an example that no sequential evaluation order can evaluate efficiently. We conclude that a parallel abstract machine is a prerequisite for space-efficient evaluation, and we suggest explicit structure-independent ways of controlling parallel

Now let us move on to a simple text processing function tp. tp allows abbreviations to be defined and substituted into the text. For simplicity, we assume that only one abbreviation need be defined at a time. An abbreviation definition appears in the text as the abbreviation enclosed in brackets, and subsequent exclamation marks are replaced by the text between the brackets. So, for example, tp would convert

```
(abbreviation)An ! definition appears in the text
as the ! enclosed in brackets.
```

into

```
An abbreviation definition appears in the text as
the abbreviation enclosed in brackets.
```

tp takes two arguments, the text to be processed (a list of characters) and the initial abbreviation value, and returns the list of characters in the result. It is defined by

```
tp [] ab        = []
tp ("!":x) ab = ab ++ tp x ab
tp ("(":x) ab = tp (after ")" x) (before ")" x)
tp (c:x) ab    = c:tp x ab
```

where before and after are assumed to return the characters in the second argument up to and beyond the first occurrence of the first argument respectively.

This observation motivates us to accumulate the sum differently, so that partial sums are reducible earlier. We introduce an accumulating parameter, thus:

```
length l = length' l 0
length' [] n = n
length' (a:x) n = length' x (n+1)
```

Now length [1,2,3] can be reduced as follows:

```
length [1,2,3] red length' [1,2,3] 0
               red length' [2,3] (0+1)
               red length' [2,3] 1
               red length' [3] (1+1)
               red length' [3] 2
               red length' [] (2+1)
               red length' [] 3
               red 3
```

using only constant space. However, this constant space version depends on the second parameter of length' being reduced before length' is applied itself. This would not be so in a completely lazy system: instead the sum would be accumulated unevaluated, as the expression (((0+1)+1)+1), and would only be reduced when returned as the result of length. So a lazy evaluator would make this version no more efficient than the former. We will show later how we can force an otherwise lazy evaluator to execute this function efficiently.

evaluation in sections 7.4 and 7.5. In sections 7.6 and 7.7 we tackle some interesting examples using our primitives, and show that they allow efficient solutions to be written. In section 7.8 we report the disadvantages of our primitives, and in section 7.9 we draw our conclusions.

7.2. LAZY EVALUATION

In this section we will explain a few examples which show that simple lazy evaluation can sometimes be grossly inefficient. First of all, we consider the function that computes the length of a list. The simplest definition we could use is the following:

```
          length [] = 0
          length (a:x) = 1 + length x
```

Although we would expect to compute the length of a list in constant space, this formulation will clearly require space proportional to the length of its argument, since, for example, length [1,2,3] is reduced as follows:

```
          length [1,2,3] red 1 + length [2,3]
                         red 1 + (1 + length [3])
                         red 1 + (1 + (1 + length []))
                         red 1 + (1 + (1 + 0))
                         ...
                         red 3
```

The chain of additions which is built up is not reducible until it is complete. When the last element of the list is scanned by length, then the entire chain must be present in store and so will consume space proportional to the length of the argument.

One would expect that the only space required by tp to process a file would be the space required to store the current abbreviation. Alas, this is not so. Consider the input list "(a)bcdefghijklmnop...". tp of this list will be reduced by a lazy evaluator as follows:

```
tp "(a)bc..." ? red tp (after ")" "a)bc...")
                         (before ")" "a)bc...")
          red tp "bc..." (before ")" "a)bc...")
          red "b":tp "cd..." (before ")"
          "a)bc...")
          ...
```

So, since the value of the abbreviation is not required until an exclamation mark occurs in the input, it will actually be stored in the unevaluated form (before ")" "a)bc..."). It will retain a pointer to almost all the input file, preventing it from being reclaimed by the garbage collector. Thus tp, like length, may require an arbitrary amount of space to run in.

In both these cases a large expression which reduces to a small value is left unevaluated for a long time, representing a gross waste of storage. In order to force earlier evaluation we introduce an optional strict function call. We define a combinator VAL by

```
VAL f x red f x
```

but we insist that x is evaluated before VAL is applied. We will usually use an infix notation, writing f (val x) instead of VAL f x, and we will feel free to use the notation (val E) in any context where E is a function argument, whether this is implicit or explicit. For example, we will happily write let x = val E in... to denote a strict declaration, and we will write [a, val b, c] to denote that b is evaluated before the tuple is constructed, while a and c are left unevaluated until required.

Now we can write

```
          length' (a:x) n = length' x (val n+1)
```

to force evaluation of length to be in the efficient order.

We can use VAL to define a "sequential evaluation" operator which evaluates its first argument and then returns its second:

```
          a ; b = (λx. b) val a
```

and we can define other functions which control evaluation order, for example force -- an identity function on lists that returns its result only after all the elements of the list have been evaluated.

```
          force l = case l of
                      [] → []
                      (a:b) → a ; force b
                     esac ; l
```

Finally we can make tp store abbreviations efficiently by writing

```
          tp ("(":x) ab = tp (after ")" x)
                              (val force (before ")" x))
```

All these evaluation control methods are in use in the Lispkit software being written by Henderson, Jones and Jones [Henderson83].

In this section we exhibited two functions which are evaluated inefficiently by a lazy evaluator and we showed how they can be made efficient using VAL. We regard this as a good solution because it does not necessitate changing program structure. It seems that we can write our programs without considering execution order at first, and then annotate them with VAL and other control functions such as force to make them efficient. Unfortunately matters are not quite that simple, as we will see in the next section.

7.3. THE NEED FOR PARALLELISM

In this section we will consider a very simple parsing problem. We want a function split which takes a list of characters and returns a pair whose first component is the first line of characters, and whose second component is the rest of the argument list. split could be defined by:

```
split l = [before nl l, after nl l]
```

where nl is the newline character. Naturally, we would like split to run in constant space if possible. For example, a program like

```
program l = let [a, b] = split l in
                   [length a, length b]
```

can obviously be executed in constant space.

However, the simple definition given above does not have this property The reason for this is (informally) that both components of the result of split contain pointers to the entire input list, and so whichever is evaluated first, the other will prevent garbage collection of any of the input. In the simple program above this will lead to at least the first line of input being present in memory in its entirety. Since the first line may be arbitrarily long, this is an intolerable overhead. (It might seem that unnecessary buffering of one line is insignificant. It should be remembered that this is an extremely simple example, and that the same behaviour can also arise in much more severe forms. For example, reading the first file off a magnetic tape is exactly analogous to the case we are discussing, but the unnecessary buffering of an entire file is much more serious).

We might try to address this problem in the same way as we solved the problems in the last section, by rewriting split in a different way and using VAL to control the execution order. However, this example does not yield to this treatment. We will prove informally that no sequential evaluator can execute any version of split efficiently.

First of all we clarify the notion of a "sequential evaluator". By this we mean that, once the evaluator has begun to reduce an expression E, it will only reduce E and other expressions that E demands until E has been completely reduced.

Now, we assume that s is a version of split which will run in constant space, provided it is used in the right context. We imagine that s is applied to the infinite list of characters from the keyboard, and that characters are typed relatively slowly. We consider the second component of s's result, the part of the input list after the first newline. This is a pointer into the input list after the first newline character, and so cannot possibly be computed until all the first line has been typed. Therefore, if the context demands this value before it has consumed all the first line, then it will be suspended until the first newline is typed, and arbitrarily much of the line may need to be "buffered up" to be consumed later. We assume, therefore, that the context consumes all of the first line before demanding the rest of the input.

Let K be the maximum number of characters of input which are held in memory simultaneously. We know that K exists, since otherwise the program would require more than constant space. Let I be the input list, and let I_n be the part of the input list after the first n characters. Since s must return its two results to the context before the context can consume any characters at all, we know that s returns before K+1 characters have been typed. Therefore the expression E denoting the "rest of the input" must be created before K+1 characters have been typed. Since E evaluates to a pointer into the input list, its unevaluated form must also contain a pointer into the input list. This unevaluated form is created before K+1 characters have been typed, and so the pointer must point before I_K. If this pointer remains unchanged throughout the consumption of the first line then it will cause most of the input to be retained in memory, consuming an arbitrary amount of space.

We must therefore assume that the reference from E to I_K is via some other expression E', which is demanded, and so reduced, by the consumption of the first line. This reduction must eliminate the pointer to I_K. We have already shown that the consumption of the first line cannot demand E, so therefore E and E' are not equal. We must have instead

$$E = F\ E'$$

for some function F. In order to satisfy our assumption about K, E' must be reduced after at most 2K characters have been typed, and consequently the result of reducing E' must contain a pointer into the input list before I_{2K}. Once again, this pointer, if retained throughout the consumption of the first line, would lead to an arbitrary amount of input being buffered in store, and so it must be via some third expression E'' which is demanded by the consumption of the first line. We must have

$$E' = F' \; E''$$

By continuing this argument, it follows that, if the first line is *n* characters long, a chain of at least n/K expressions will be built up. Since none of the expressions are equal, at least n/K cells will be used, and a non-constant amount of space will be consumed. This proves our assertion: no sequential evaluator can execute any version of split in constant space.

Let us consider the significance of this result. We have demonstrated only that one particular useful function cannot be run efficiently on sequential abstract machines: however, it is intuitively clear that a similar problem arises whenever there is more than one "consumer" for a value: if the sequential nature of the abstract machine forces one consumer to run first then the other will retain the value and delay garbage collection. (In the case of split, there are two consumers. They are its two results). Even if it were possible to combine the two consumers into one this would be a bad solution since it would destroy the modularity of the program. We have shown that even this bad solution is not possible in general on a sequential machine. We are therefore compelled to introduce parallelism into our abstract machine.

Having done so the problem is not yet solved: we must address the problem of scheduling reductions so as to minimise the space used. We can take two approaches here: either scheduling is implicitly performed by the implementation, or it is explicitly controlled by the programmer. Some progress has been made with the former approach. Wadler has shown in [Wadler83] that if a program can be executed in constant space then all scheduling can be done at compile time, and he has given an algorithm for doing so. However, he has also shown that, if a program cannot be executed in constant space, then the problem of scheduling its execution to minimise the space used is NP-complete [Wadler83]. Although implicit scheduling seems the more desirable solution in the long run, we consider that these difficulties make it impractical at present. We prefer to look for simple ways in which the programmer can control scheduling explicitly.

7.4. PAR FOR MORE PARALLELISM

We begin by providing an explicit mechanism for starting parallel evaluation. By analogy with the VAL function which we introduced in section 7.2, we define a combinator PAR by

```
        PAR f x red f x
```

with the additional property that x starts executing in parallel before the reduction is performed. Since PAR does not wait for x to finish it is truly an identity function (VAL is not because it is strict in x). We will use an infix notation for PAR and define control functions for use with it, just as we did for VAL.

First we note that the *split* problem can be solved by defining split by

```
split l = [before nl l, par after nl l]
```

so that, as the context consumes the first line, the second component is also executing and swallows characters as they arrive. (par is the infix version of PAR, and the notation in this example means that (after nl l) starts executing immediately the result of split is constructed).

Now, if we define

```
a || b = (λx. b) par a
```

which starts a executing in parallel and returns b, then we can write

```
parlist l = case l of
              [] → []
              (a:b) → a || parlist b
            esac || l
```

parlist l returns l, but starts parallel evaluation of all l's components. We could use parlist in the definition of split thus:

```
split l = [par parlist (before nl l),
                par after nl l]
```

which is better than the former in that, even if the context does not refer to the first line until long after it has consumed the rest of the input, the first line will be computed and will not retain a pointer to the entire input list. So the programmer is able to ensure that split runs efficiently by annotating it with par and control functions.

7.5. SYNCH FOR LESS PARALLELISM

We have shown how the programmer can deliberately introduce parallelism to make split run efficiently when applied to the keyboard, but we have so far overlooked the need to reduce parallelism by synchronisation. To see why this can be necessary, reconsider our last definition of split:

```
split l = [par parlist (before nl l),
               par after nl l]
```

If split is not applied to the keyboard, but to some computed list, then there is a danger that the expression (par after nl l) will cause l to be evaluated faster than the context is able to consume the first line. This may result in much of the first line being buffered while the context catches up. We need to synchronise the consumption of l in the two expressions.

We recall at this stage that "lazy evaluation" scheduling of graph reduction is achieved by propagation of demand. Originally the result of the whole program is demanded, and thereafter demand is propagated by strict operations through the graph. Only parts of the graph at which demand has arrived are reduced. PAR modifies the propagation of demand by propagating it to two nodes simulteneously. We now need to synchronise execution by restraining the propagation of demand.

We introduce a new function SYNCH to achieve this. SYNCH is defined by

```
SYNCH e = [e, e]
```

However, the two copies of e which are returned are actually different: call them e_1 and e_2. No demand is propagated from e_1 or e_2 to e until both have been demanded. Thus, if we write

```
let [a, b] = SYNCH (1 + 2) in
     [par factorial a, par fibonacci b]
```

we can be sure that 1 and 2 will not be added until both factorial and fibonacci are ready to use the answer. SYNCH is actually a dangerous function in that it can cause deadlock if one of e_1 and e_2 is never demanded. Nevertheless, it is an important control mechanism.

In the split example, we need to ensure that two separate processes consume a list at the same rate. We will define a function SYNCHLIST that takes a list and returns two versions of it, in such a way that both versions must be consumed at the same rate.

```
SYNCHLIST l =
     let [s1, s2] = SYNCHLIST (tail l) in
     let [l1, l2] = case l of
                      [] → [[], []]
                      (x:l') → [x:s1, x:s2]
                     esac in
     let [w1, w2] = SYNCH l in
         [w1 ; l1, w2 ; l2]
```

SYNCHLIST works as follows: when it is first called none of the declarations are evaluated and it returns its two results immediately. Eventually, demand will arrive at one of them, say (w1: l1), and so w1 will be demanded. Since w1 is a result from SYNCH the computation will be suspended at this point. Later another parallel process will demand (w2: l2), and so demand will arrive at w2. Now the conditions for SYNCH to propagate demand are satisfied, and so SYNCH's argument l will be evaluated and returned as the value of w1 and w2. This will allow l1 and l2 to begin executing, and so each result of SYNCHLIST will be computed, giving x:<another synchronised list>. Thus, as required, SYNCHLIST constrains the consumers of its two results to work at the same rate.

Now if we define

```
split l = let [l1, l2] = SYNCHLIST l in
                  [before nl l1, par after nl l2]
```

then we are assured that characters will be consumed by both processes as they are demanded by the consumer. In this case it would be inappropriate to apply parlist to the first result since we want evaluation to be driven by the consumer of the first line. (Technical point: for the purposes of SYNCH we regard demand to have arrived at an expression when it is reclaimed by the garbage collector. Without this the example above would deadlock at the end of the first line because no more demands would arrive at l1).

So by annotating split with explicit parallelism and synchronisation functions we can make it run in constant space in a variety of contexts. This annotation does not represent a change in the structure of the program, and so we really can write the program first, without regard to evaluation order, and annotate it afterwards. However, we must take into account the intended use of a function when we annotate it, and we may need differently annotated versions for use in different contexts. In the remainder of this chapter we will give a few more examples of the application of PAR and SYNCH.

7.6. QUICKSORT

Quicksort can be expressed very elegantly in a functional language. It can be defined by the equations

```
sort [] = []
sort (a:x) = sort {b<-x; b<a} ++
             [a] ++
             sort {b<-x; b>a}
```

using Turner's *ZF* notation [Turner81] (*{b←x; b<a}* means the list of elements b of x which are less than a). The imperative Quicksort sorts a list of n elements in space O(n) and time $O(n^2)$ in the worst case, O(nlog*n*) on average. We shall examine the complexity of the functional version.

First of all we consider the time taken to sort a list of n elements, T(n). On average, the two recursive calls to sort will be on lists of length n/2, and so we have

```
T(0) = a constant
T(n) = 2T(n/2) + the rest of the cost
```

Since the rest of the cost of a call of sort is proportional to the length of the argument, we have

$$T(n) = 2T(n/2) + O(n)$$

which has the solution

$$T(n) = O(n\log n)$$

However, in the worst case the list will be split into an empty list and a list of length n-1, and so we will have

$$T(n) = T(n-1) + O(n)$$

with the solution

$$T(n) = O(n^2)$$

So we observe with satisfaction that the functional version of Quicksort has the same time complexity as the imperative one.

Now, let us consider the space complexity, S(n). Clearly $S(n) \leq T(n)$. We consider the amount of space in use just after computing the first element of the sorted list. Since this element could originally be anywhere in the argument to sort, it is clear that sort must force the complete evaluation of its argument before it can compute this element. So, considering the second equation for sort, we see that by this stage the expression $(b \leftarrow x;\ b<a)$ must be completely evaluated, and so will not share any cells with the original argument. On a lazy evaluator, the second recursive call of sort,

sort (b←x; b>a) will not have been evaluated at all, and so will contain a reference to the original argument x Therefore the space in use will be approximately the space required for x plus the space required to sort (b←x, b<a) In the average case, then.

$$S(n) = n + S(n/2)$$

and so

$$S(n) = O(n)$$

In the worst case, however,

$$S(n) = n + S(n-1)$$

and so

$$S(n) = O(n^2)$$

So the worst case space complexity of the functional Quicksort is as bad as its time complexity, far worse than the imperative equivalent. Even in cases that deviate slightly from the average, this means a worse space behaviour.

Of course, this is just the kind of problem that our primitives were intended to solve. We shall use them to force both recursive sorts to consume the argument list at the same rate. This can be done by defining

```
sort (a:x) = let [x1, x2] = SYNCHLIST x in
                    par sort {b←x1; b<a} ++
                    par [a] ++
                    par sort {b←x2; b>a}
```

To find the space complexity of this function, we observe that, since access to the argument list is synchronised, it is consumed as it is computed. No storage is wasted in buffering. Therefore, each recursive invocation of sort requires only a constant amount of space. Also, the number of parallel invocations of sort is bounded by twice the number of elements to be sorted, since an element is consumed when two parallel invocations are started. It follows that this sort function requires only O(n) space to sort n elements, as good a result as the imperative one. Moreover, on a machine with enough parallelism, this version of sort will sort n elements in O(n) time using (on average) O(logn) processors.

This example shows that our primitives can be used not only to make certain programs run in constant space, but can also provide substantial improvements in the space complexity of programs that do not.

7.7. PIPES

The UNIX operating system provides a facility whereby two (or more) programs may be run in parallel connected by a "pipe", so that one program receives as input the output of the other. This can be a convenient way of writing multi-pass compilers, for example, so that each pass receives the output of the previous one, and all passes can run in parallel and exploit the capabilities of multi-processor machines. The programs are loosely synchronised in that each producer may run ahead of its consumer, but only by a limited amount.

Pipes are naturally incorporated into a functional operating system. Assuming that programs are functions from a list of inputs to a list of outputs, we may connect two programs using the function

```
connect prog1 prog2 input = prog1 (prog2 input)
```

which just composes the two programs. Running (connect prog1 prog2) effectively runs prog1 and prog2 in parallel, connected by a pipe. However, in this case the programs are synchronised exactly, since prog2 will only run when prog1 demands an input. This is undesirable if the underlying machine is capable of real parallelism, because it could lead to processors standing idle while there is real work to be done. We can easily correct this by defining

```
connect prog1 prog2 input =
    prog1 par pipe N (prog2 input)
N = some number
```

using the function pipe which takes a number and a list and returns a copy of the list, but starts the evaluation of the N+I'th list element as soon as the I'th is demanded. pipe is defined by

```
pipe n l = parlist (take n l) ||
           pipe' (drop n l) l
pipe' [] l = l
pipe' (a:f) (b:l) = a || (b:pipe' f l)
```

Now when (connect prog1 prog2) is run, prog2 is started immediately and its first N outputs are demanded. Thereafter, whenever prog1 consumes one of its inputs, another output of prog2 starts to be computed. prog1 and prog2 run in parallel and prog2 is allowed to get up to N elements ahead. This is exactly the behaviour of two programs connected by a pipe in UNIX

The method we have used here is actually more general than the UNIX pipe. Although we have applied it to lists, we could as easily write a function which works on trees, or any other data-structure. We have presented a general way of starting the evaluation of data shortly before it is required; it can be used in any context where this is desirable, for example to start disc transfers shortly before the data on the disc is needed, or just to increase the available parallelism in a controlled way.

7.8. DISADVANTAGES

Since PAR and SYNCH are a control mechanism, not a panacea, it is possible to use them to cause undesirable consequences. There are two main categories of such errors: deadlock and rampant parallelism. Deadlock occurs if SYNCH is used without a corresponding PAR. For example, if sort were defined by

```
sort (a:x) = let [x1, x2] = SYNCHLIST x in
                    sort [b←x1; b<a] ++
                    [a] ++
                    sort [b←x2; b>a]
```

then it would never produce any results because demand would never arrive at x2. On the other hand, rampant parallelism occurs if PAR is used too often without corresponding SYNCHs. For example, if pipe were defined by

```
pipe n l = parlist l
```

then it would try to compute all the output of the producer process in parallel, possibly clogging the system.

Thesa problems are worrying because a good understanding of the implementation is required to avoid them. The naive user is unlikely to see anything wrong with the erroneous definitions in this section. In fact, it is necessary to know tricky details of the implementation in order to predict how some programs will behave. For example,

```
let [a, b] = SYNCH e in a + b
```

will be evaluated correctly if the implementation evaluates the arguments of + in parallel, but will deadlock if they are evaluated in sequence. Another example is the function exists

```
exists p = p, \ function p
exists p = exiets (p true) or exists (p false)
```

which takes a predicate, a curried function expecting several boolean values and returning a boolean, and returns true if its argument is not identically false. If the arguments of or are evaluated in sequence then exists performs a space efficient sequential search, but if they are evaluated in parallel then rampant parallelism will result which will clog the system completely.

Therefore PAR and SYNCH, although flexible, require considerable expertise in their use. This may signify that they should be replaced by a more structured equivalent; but at the moment we ere unable to suggest what form it might take.

7.9. CONCLUSION

In this chapter we have demonstrated beyond doubt that parallel abstract machines are a prerequisite for efficient implementations of functional languages, and that correct scheduling is vital if this efficiency is to be achieved. It is debatable which is the best way to ensure correct scheduling, but we favour explicit control by the programmer using simple primitives. We have defined two such primitives and worked enough examples to engender confidence that they are sufficiently powerful for most practical problems. Our primitives have the important advantage that they are structure-independent, so that the programmer does not have to take them into consideration when first designing his program.

CHAPTER 8

GARBAGE COLLECTION

8.1. INTRODUCTION

We have argued in chapter 2 that one of the important advantages of functional languages is the provision of a garbage collector. Following Dennis [Dennis81] we also believe that the best way to provide filestores and databases is within a very large garbage-collected virtual memory. In this chapter we concern ourselves with garbage collection methods suitable for functional languages running in very large virtual memories.

In section 8.2 we will discuss the general strategies available and argue that reference counting is the most promising for our needs. In section 8.3 we will examine how circular structures, the bane of reference counting garbage collectors, arise. In sections 8.4 and 8.5 we will present our extension to reference counting for managing circular structures. Finally, in section 8.6 we will examine the costs of our method. In section 8.7 we will refer briefly to Brownbridge's method, and in section 8.8 we will present our conclusions.

8.2. GARBAGE COLLECTION STRATEGIES

There are two main strategies for deleting objects no longer required, which we shall refer to as mark-scan garbage collection and reference counting garbage collection. Mark-scan algorithms determine which objects can be deleted by visiting all objects accessible from any machine register, and then deleting the others. Reference counting algorithms store a count of the number of references to an object with the object, and delete the object when this count becomes zero. Both categories cover a wide variety of algorithms, and, in particular, either strategy may be used in parallel with the main computation [Dijkstra79] [Hudak82] [Grit81].

In the context we have described, however, reference counting seems to have a marked advantage. Firstly, visiting every accessible object, as mark-scan algorithms do, is very expensive when there are a very large number of objects. Our very large virtual memory will contain a very large number of accessible objects. Reference counting algorithms, on the other hand, visit only objects as they are being processed.

Secondly, merk-scan algorithms may not delete inaccessible objects until long after they become inaccessible, while reference counting algorithms delete them immediately. This is particulerly important in a virtual memory system, because it means that, with reference counting, short-lived objects can all share the same locations. These locations will therefore form part of the computation's working set. Since applicative language execution generates a very large number of short-lived objects, we do not believe that a very large virtual memory is viable if they are not deleted promptly

Thirdly, mark-scan algorithms operate in several phases, and some kind of synchronisation is necessary when the phase changes. This is, at the least, an inconvenience. In contrast, reference counting involves only very local information and requires no global synchronisation.

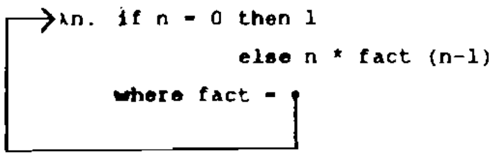
For these reasons we do not believe that mark-scan garbage collection is viable in largo virtual memory parallel applicative systems. However, reference counting also suffers from a serious disadvantage – objects which are accessible from themselves, and so form part of circular structures, are not deleted at all. Since even the ubiquitous recursive function is usually represented by a circular structure, this presents a very serious problem which could negate all the other advantages of reference counting.

6.3. CIRCULARITY

It is worth examining how circular structures arise during execution. Some are created becausa the programmer has deliberately defined a circular data-structure, for example by

```
        let x = cons (1, x)
```

The majority are the representations of recursive functions. For example, the factorial function might be represented by



```
λn. if n = 0 then 1
               else n * fact (n-1)
      where fact = •
```

where the circular pointer is shown by the arrow.

It has been suggested that it is not necessary to use circular structures other than recursive functions (which are vital since iteration is expressed as recursion in applicative languages). Dennis has proposed a non-circular representation for recursive functions in [Dennis82], and Friedman and Wise have observed in [Friedman79] that reference counting will work for recursive functions (in their representation) if the circular pointer is not counted.

However, neither of these schemes allows the construction of genuinely circular data-structures. Our own experience of using KRC [Turner81], which does not allow circular structures to be constructed at run-time, suggests that they are almost always unnecessary; but occasionally a program cannot be written efficiently and conveniently without them.

An example of such a program is a programming language interpreter. The problem here is that, since functions in the language being interpreted must be represented by data-structures in the underlying language, recursive funcions would naturally be represented by circular data-structures. We have encountered similer problems in writing a synchronous process simulator and in the algorithm for compilation to super-combinators given in section 5.4. In these cases it seems to be either very difficult or extremely inefficient to program without circular structures.

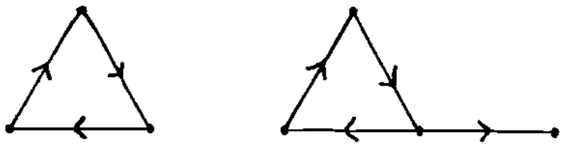
Bobrow has described [Bobrow80] a method that manages circular data-structures using information supplied by the programmer. His method has some strong similarities to our own; however, changes to the data-structures can invalidate the information the programmer supplied, leading to delay or outright failure in reclaiming some objects. For this reason its applicability is limited.

We believe that the extension of reference counting to deal with circular structures in all their generality is essential.

8.4. THE STATIC PROGRAM GRAPH

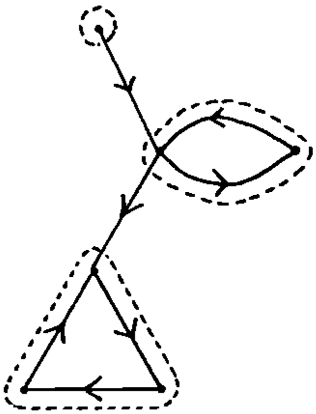
We restrict ourselves to super-combinator implementations of functional languages, and for the time being we assume that the graphical combinators introduced in section 4.5 are not used. We begin our extension of reference counting by considering a snapshot of the program graph at a particular instant, and looking for a way to describe its circularities. We recall two definitions from graph theory.

Definition: A graph is *strongly connected* if, for any two nodes A and B, there is a path from A to B and vice versa. For example, the left hand graph below is strongly connected, but the right hand graph is not.



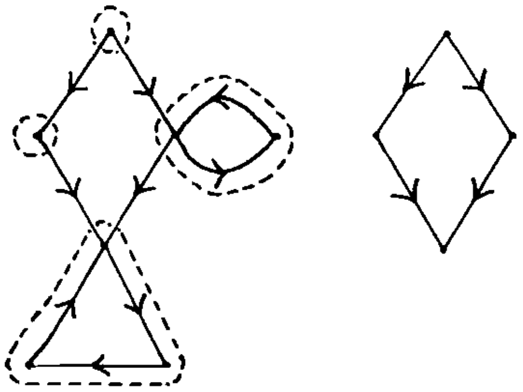
Definition: A *strongly connected component* of a graph is a maximal strongly connected subgraph.

It follows from these definitions that any graph can be decomposed into disjoint strongly connected components. The figure below shows a sample graph and its decomposition.



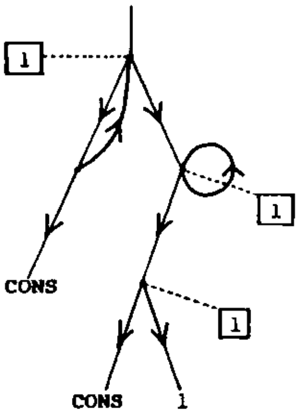
The strongly connected components of a program graph are the units of circularity: for, if one node in a component is accessible, then the whole component is accessible; conversely, if one node in a component can be deleted, then all the nodes in the component can. It follows that it is more appropriate to consider garbage collection of strongly connected components than of individual nodes.

To this end we define the *derived graph* G' of a graph G to be the result of coalescing all nodes in the same strongly connected component. More precisely, the nodes of G' are the strongly connected components of G, and there is an edge from node A to B of G' if there is an edge in G from a node in A to a node in B. The next diagram shows an example graph and its derived graph.



The derived graph is always acyclic. We can see this because, if it were not, then there would be two distinct strongly connected components A and B with paths both from A to B and from B to A. But this means that, in the original graph, there must have been paths from a node in A to a node in B and vice versa. Because A and B are both strongly connected, this implies that there must have been a path from any node in A to any node in B and vice versa; so A and B would both be part of the same strongly connected component. This violates our assumption that A and B were disjoint.

Since the derived graph is acyclic, it can be garbage collected safely using reference counting. This motivates us to modify the program graph by adding a field to each node which points to a shared reference count. All the nodes in the same strongly connected component point to the same shared reference count, which contains the number of references to the whole component from other components. An example of a program graph with shared reference counts drawn as boxed numbers is



We can now delete a whole strongly connected component when the shared reference count becomes zero. We can also perform a fast test to discover whether two nodes are in the same strongly connected component, by comparing their shared reference count pointers. This will be important later.

As well as the shared reference counts, we retain the reference counts of each individual node, which we refer to as the local reference counts.

6.5. THE GRAPH IN MOTION

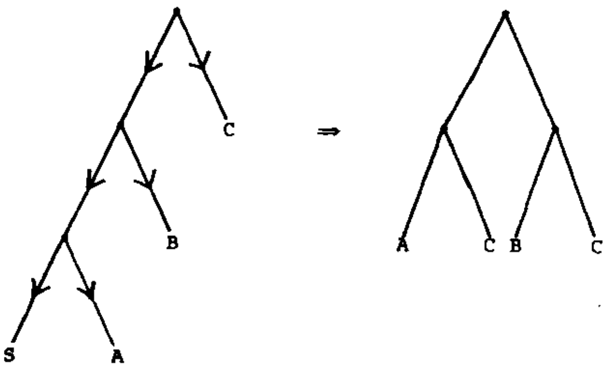
We can now use reference counting to reclaim circular structures provided we can keep the shared reference count structure up to date. In this section we show how the machine can update the shared reference counts during reduction.

Since our machine language is applicative, all the reduction rules obey a very important property: all the nodes accessible from the root of a reduction after the reduction are either newly claimed, or were already accessible before the reduction. Because of this, two distinct strongly connected components will always remain so, since no reduction can make one accessible from the other if it was not already so. The only ways in which a strongly connected component can change are by growing, *ie* having newly created nodes added to it, or by splitting into several smaller ones.

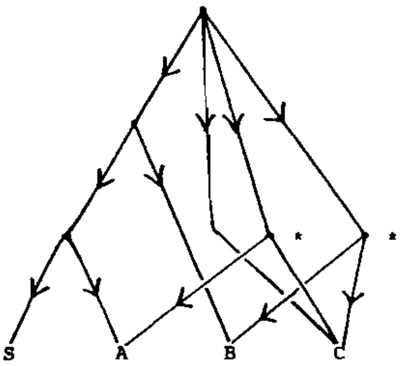
For the purposes of exposition, we consider each reduction to happen in two stages. In the first stage, new cells are claimed and pointers to the result are added to the node being reduced. This may cause strongly connected components to grow, but cannot cause them to split. In the second stage, the old pointers from the root are deleted. This may cause strongly connected components to split, but cannot cause them to grow. In between the two stages there will be more than two pointers from the root. This will cause no problems in the implementation because some of the pointers will actually be held in registers.

8.5.1. ADDING NODES AND POINTERS

When adding nodes and pointers to the program graph we must decide which strongly connected component the new nodes belong to. Having decided this the local and shared reference counts may be adjusted accordingly. We illustrate how this may be done using the combinator S, whose effect is shown below.



After the first stage in the application of S, the graph will appear as shown below, with four pointers from the root. The nodes marked * are new and must be allocated to a strongly connected component.

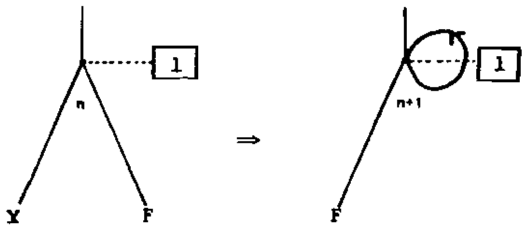


We notice the following property of strongly connected components: if a node is part of a non-trivial component (one consisting of more than one node) then at least one of the nodes pointing to it is in the same component. Since the starred nodes form a tree, there is only one node pointing at each one: the node directly above it in the tree. It follows that if any starred node is in a non-trivial component then the node directly above it is in the same component, and, ultimately, so is the root. Therefore, every starred node either forms a trivial strongly connected component by itself, or is in the same component as the root.

In order to decide which of these cases applies, we notice another property of strongly connected components: if a node is part of a non-trivial component then at least one of the nodes it points to is part of the same component. A starred node in a non-trivial component must therefore point either to an argument of the reduction in the same component, or to another starred component with the same property. It follows that all starred nodes in non-trivial components lie on paths from the root to an argument in the same component.

Conversely, any node on a path between two nodes in the same component is also in that component. We may therefore allocate new nodes to components as follows: when a reduction is performed then any new nodes on a path from the root to an argument in the same strongly connected component as the root should be added to that component. Others should form new, trivial components by themselves. This is a simple and complete rule for keeping our data-structures correct.

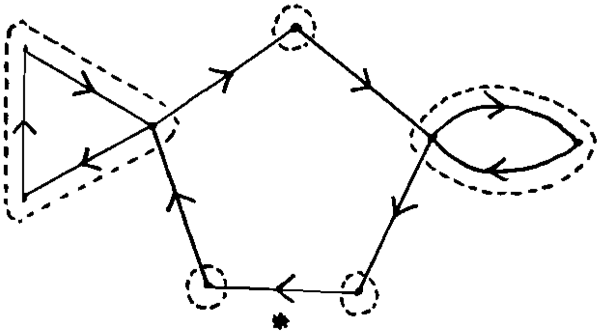
Even the Y operator is covered by this rule. It adds no new nodes to the graph, but adds a circular pointer. This pointer cannot alter the strongly connected component structure of the graph: it only alters the local reference count of the root.



8.5.2. DELETING POINTERS

When a pointer is deleted there are two cases to consider. If the pointer connects nodes in two different strongly connected components then it is only necessary to update the local and shared reference counts of the target, and delete the target if necessary. All the nodes of a strongly connected component can be deleted at once because they are all accessible from any node in the component.

If the pointer connects two nodes in the same strongly connected component, then it is possible that the component may split into many smaller ones. For example, deleting the marked pointer in the diagram below splits the strongly connected component up as shown.



We have to find the strongly connected components of the component being split, and mark them as new strongly connected components of the program graph. There is an efficient algorithm due to Tarjan [Tarjan72] for finding the strongly connected components of a graph in one scan over it. We can apply this algorithm to the component being split, and it will run quickly because we are only applying it to a small part of the program graph. At the same time we can compute the shared reference counts of the new components by adding the local reference counts of their constituent nodes together and subtracting the number of internal pointers. Any component whose shared reference count is zero is deleted.

8.8. EXAMINATION OF COSTS

This method requires more storage then traditional reference counting, because each node must have a shared reference count pointer stored in it. Since, in general, a reference count requires about as many bits as a pointer, it would seem that adding a shared reference count pointer to a node already containing a local reference count, a head pointer, and a tail pointer, represents about a 33% increase in storage requirement. In practice we expect the increase to be smaller than this, because nodes will also need space for scheduling information and the like (nodes in ALICE [Darlington81] occupy a total of 32 bytes).

Very little extra store need be used to hold the shared reference counts themselves. If one of the nodes of each strongly connected component is distinguished in some way, then the shared reference count can be stored in its shared reference count pointer field, and the other nodes in the component can point to it. This requires only one bit per node. Using this scheme non-circular parts of the graph, which consist of strongly connected components of one node each, can be stored compactly and have their reference counts updated quickly. This is very important because the bulk of the graph is of this form.

Our method is also slightly slower than ordinary reference counting, partly because of the extra cost of updating shared reference counts, but mainly because of the cost of splitting up strongly connected components. Fortunately, we do not believe strongly connected components will be split very often. This is because circular structures are usually built and then used several times: for example recursive functions are usually called many times. Since strongly connected components need to be split only when they change shape, this will happen while they are being built, but not while they are in use. Therefore, for example, calling a recursive function should involve no splitting.

The amount of splitting could be reduced still further by using graphical combinators, as suggested in section 4.5. This would allow many circular structures to be built in one step, rather than by applying Y followed by tree combinators, and so would eliminate splitting in these cases. Provided that the combinator body contains information on its own strongly connected component structure, the structure of the result of application is easy to determine, in a similar way to that used above.

Our method shares the advantages of ordinary reference counting, namely that reference counting activities can be done in parallel with the main computation, and that only local information is required. We therefore expect it to be suitable for virtual memory applicative systems, and reasonably efficient.

8.7. BROWNBRIDGE'S METHOD

Our method differs from the other approaches we discussed in section 8.3 in being a complete solution to the garbage collection problem for graph reduction implementations: that is, no additional restrictions need be placed on the applicative programmer to ensure that garbage collection works. Brownbridge will shortly publish an extension of reference counting that is completely general [Brownbridge83]. His method distinguishes 'strong' and 'weak' pointers, and reference counts the two kinds separately. When the last strong pointer to a node is deleted the garbage collector alters the status of enough other pointers to ensure that the whole graph is spanned by an acyclic graph of strong pointers, and deletes parts of the graph that are no longer referenced. On occasion this process may need to scan a large part of the graph.

The great advantage of Brownbridge's method over our own is that it requires no assumptions about the kind of use to which it is put, whereas our method can only be used with graph reduction implementations of applicative languages. However, since the two algorithms are so different, it is very difficult to predict which is the more efficient for any particular use to which they are both applicable. We are unable to compare them in any more detail since we do not understand Brownbridge's algorithm yet.

6.8. CONCLUSION

We expect that computer systems in the future will be highly parallel, and will have large virtual memories. Garbage collection will be essential, but mark-scan garbage collection will be impractical. In the past, reference counting garbage collection has been unable to collect general circular structures, although particular cases have been covered. We believe that such general circular structures are a vital programming tool. We have shown how, at least in a graph-reduction machine executing applicative programs, reference counting can be extended to handle any structure at all. Our method is slightly more costly than ordinary reference counting, but, we believe, vastly cheaper than excluding circular structures altogether.

CHAPTER 9

RELATED WORK

9.1. INTRODUCTION

In this chapter we survey other work in the area of the efficient implementation of functional languages. The field is very active, thanks partly to the software engineering advantages of functional languages, and partly to the promise that they can be used to exploit highly parallel architectures. As a consequence, almost all possible avenues are being investigated. In section 9.2 we will describe two string reduction architectures. In section 9.3 we will consider architectures that perform graph reduction on the original program, or some not-very-heavily compiled version of it. Section 9.4 describes several approaches based on the SECD machine. In section 9.5 we will discuss attempts to exploit Turner's SKI approach fairly directly, and in section 9.6 we will describe implementations based on super-combinators or very similar ideas. Finally, section 9.7 concerns itself with dataflow.

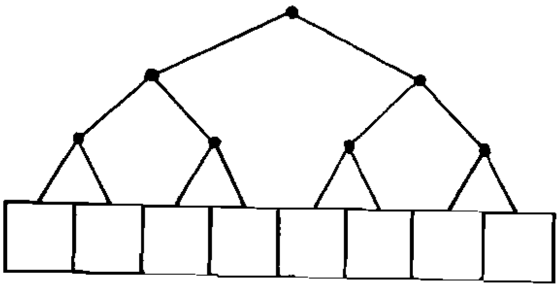
9.2. STRING REDUCTION

In this section we will describe two string reduction architectures, one designed by Berkling and one by Mago.

Berkling's architecture [Berkling] is designed for reducing λ-expressions. It consists of a processor and two stacks, a left stack and a right stack (in fact the processor contains several other stacks for holding expressions temporarily). The machine starts with the input expression in either the left or the right stack and transfers it back and forth between them performing β reductions until it is no longer reducible. The final expression is then output. A prototype machine was actually constructed.

Although this is perhaps the most natural way to construct a reduction machine, it has a number of disadvantages. Firstly, since no attempt is made to select appropriate expressions for reduction, the machine can waste a great deal of time on reductions which are not actually necessary. This is compounded by the substitution of partially reduced arguments into function bodies: if the argument is used more than once then any subsequent reductions of it must be duplicated. Secondly, operations on large data-structures are expensive since the whole data-structure must be scanned for every operation. However, the architecture can easily be extended to exploit several processors, and this might partially outweigh the disadvantages.

Mago's architecture is entirely different. It is designed to execute Backus' FP [Backus78] rather than the λ-calculus. The machine consists of a large number of cells, each of one of two kinds. One kind of cell is arranged in a long linear sequence which holds the symbols of the FP program being reduced, and the other kind is arranged as a binary tree connected to the first kind at the leaves.



The machine operates in cycles, during which all reducible expressions are located, microcode broadcast through the tree to the cells that need it, empty spaces rearranged to make room for expansion of the program, and finally, the reductions performed. The machine is therefore able to reduce all reducible expressions in the program at the same time, provided there is space to hold all the results.

Mago's machine circumvents the first flaw in Berkling's. An expression is only considered reducible if all its sub-expressions are completely reduced. This ensures that no reducible expression is ever copied, and so work is never duplicated. The price, however, is that all functions are strict, and so the programmer cannot take advantage of lazy evaluation. Since we regard lazy evaluation as one of the most important advantages of functional programming, we consider this a serious deficiency.

The second flaw in Berkling's machine is also present in Mago's: since arguments are copied from place to place, it is very expensive to manipulate large data-structures. Mago has proposed a partial solution to this problem

[Mego81], whereby the machine can be made to leave a large data-structure where it is and move other arguments and results past it. This is not a completely general solution, however. It remains to be seen to what extent the advantages of parallelism in this machine will be offset by the cost of data movement.

9.3. GRAPH REDUCTION OF THE SOURCE

Several people have designed implementations which (more or less) perform graph reduction on the original source of the program. We say "more or less" because many of these implementations perform some trivial compilation, but nothing so major as compiling to Turner's combinators or to SECD machine code.

Turner has used graph reduction of the source in his KRC implementation [Turner81]. This is an interpretive implementation on conventional machines, which uses the KRC program, a flat set of recursion equations, as the reduction rules for the interpreter. The advantage on a conventional machine is that it is able to produce error messages that are extremely intelligible to the programmer, since they contain references to names and expressions in the original source.

Keller, Lindstrom and Patil use graph reduction of source programs in their AMPS (applicative multiprocessing system) [Keller79]. In their case the machine is programmed in a "functional graph language" (FGL), a dialect of LISP, and the machine operates by graph reduction of FGL programs. Like Mego's machine, their design consists of a large number of parallel processors connected to the leaves of a binary tree of different processors. However, in their case the individual processors are each capable of significant computations, and the tree serves only for communication and load balancing. The communication tree supports a global address space, so any process may be run on any processor. Processes are created by INVOKE instructions, analogous to subroutine calls. The AMPS does not try to exploit parallelism on a very small scale (within subroutines). The INVOKE instruction places the newly created process on an INVOKE-list in the executing processor, and when the processor finishes (or suspends) the current process it takes the next one from the INVOKE-list and proceeds with that instead. Periodically the communication tree obtains the length of the INVOKE-list in each processor and transfers processes from heavily loaded processors to nearby lightly loaded ones. In this way work is distributed through the machine.

The AMPS communication tree is able to support local communications very efficiently, since they need only pass through a small number of nodes. This is both its strength and its weakness. Its designers believe that executing programs will exhibit sufficient locality of reference that almost all communications will be local and therefore fast. However, since all communications travelling a distance of half the machine or more must go through the root of the tree, then if only a small percentage of communications turn out to be truly long-distance the root will become the bottleneck of the entire system.

ALICE [Darlington81] is another design using graph reduction of the source. In this case the processors and memories are separate, and communicate through a packet switching network. Thus all processors are the same distance from all memories, and locality of reference is not an issue. The program graph consists of applications of functions whose definitions are held in the microcode store. This store must be loaded with definitions of all functions in the user's program before execution begins. ALICE also supports λ-expressions, which it implements by copying the function body substituting for the arguments, so nested λ-expressions are not very efficient. Only reference counting garbage collection is performed by the hardware, so additional garbage collection must be programmed explicitly if circular structures are to be used. Reducible nodes are held in a pool which circulates among the processors, and so when a processor becomes idle it need only grab one of these nodes to continue executing. The grain of parallelism is very fine, since all the operations in a function may be performed in parallel. Parallelism can be controlled explicitly, and, in particular, both PAR and SYNCH (chapter 7) can easily be implemented

ALICE contains some messy features, for example, uncurried functions with at most three arguments are much more efficient than other kinds. We suspect that efforts to exploit such features will lead to complex end unwieldy software Nevertheless, the overall design accords very closely with our own views, and we are confident that all the results of our work would prove easily applicable to ALICE.

9.4. SECD MACHINES

The SECD machine (described in section 3.4) has also been used as the basis for functional language implementations. Henderson uses this approach in his implementations of Lispkit, a very simple dialect of Lisp [Henderson80]. He compiles Lispkit into an (almost) linear machine code with 21 instructions, and then executes the machine code with an interpreter. Interpreters have been written for a wide variety of machines (ranging from a Z80 to a VAX) and Henderson is now considering a hardware implementation. A considerable body of Lispkit software has been written by Henderson, Jones and Jones [Henderson83], including a complete self-hosting programming environment.

These implemantations perform fairly well, but have a tendency to run out of store during long computations. This is partly due to the problems discussed in chapter 7, since the implementations are completely sequential. It is also due to the use of environments, which by their very nature are wasteful of store, since an environment may be retained because one value in it is needed, and may thus cause the retention of many objects which could actually be thrown away. (This problem could be ameliorated by using free variable lists rather than environments, which contain only the part of the environment that is actually needed. However, many more free variable lists than environments would need to be constructed, and so the gains from this technique might well be outweighed by the extra cost of construction). Nevertheless Lispkit is a practical and proven programming system.

Steele and Sussman also used the SECD machine as the basis for their VLSI Lisp interpreter, SCHEME-79 [Steele80]. Their processor interprets a dialect of Lisp called SCHEME [Steele78], which is similar to Lispkit Lisp. The most important differences are that SCHEME is not purely functional, and that it does not support lazy evaluation. A very simple compiler converts standard functions in SCHEME programs into indexes into the microcode, and variable names into indexes into the environment. The SCHEME-79 processor is able to interpret Lisp at about the same speed as a PDP-10 KA-10, and Steele and Sussman expect an improved version to be an order of magnitude faster.

Like Henderson's Lispkit implementation, SCHEME makes no attempt to exploit parallel processors. It is also subject to the same criticism that environments are wasteful of storage, although this matters less in the absence of lazy evaluation since suspended expressions, the main sources of references to environments, do not occur. Its advantage is that it involves fairly well-understood technology, and so it is more practical in the short term than some of the other approaches in this chapter.

The SECD machine is also at the heart of a project involving Friedman, Wise, Johnson and Kohlstaedt, whose aim is to design a parallel applicative language processor. Their work is based on a language called Daisy [Kohlstaedt81] which runs on a virtual machine called DSI [Johnson81]. DSI provides lazy evaluation, a non-deterministic multi-set constructor for initiating parallelism [Friedman80], and allows some circular structures to be created. An extension of reference counting garbage collection due to Friedman and Wise is used which is able to recover the permissible circular structures [Friedman79]. At present DSI is implemented by an interpreter running on a conventional machine, but it is expected that a parallel implementation in hardware will be constructed eventually. This implementation will make use of many processors and many memories connected by a switching network described in [Wise81], an architecture quite similar to ALICE.

Since DSI is an SECD machine the same criticisms of the inefficiency of using environments apply to it. However, Daisy includes some features designed to minimise the unnecessary retention of environments, and DSI does permit parallelism, so the problem may be much less severe in this context. Parallelism is controlled in a very different way from our own suggestions, and it is not clear whether the programmer can control store use in Daisy as effectively as he can using PAR and SYNCH.

9.5. TURNER'S COMBINATORS

We have already given a very brief explanation of Turner's combinator implementation technique [Turner79] in section 3.6. We remarked there that in his implementation of SASL, Turner used some additional combinators. We will not present them in detail, but we will explain their flavour. Recall that S is defined by

$$S = \lambda abc.\ a\ c\ (b\ c)$$

and introduced into compiled programs by the rule

$$\lambda V.E_1\ E_2\ \mathbf{trans}\ S\ (\lambda V.\ E_1)\ (\lambda V.\ E_2)$$

S may be regarded as a 'director' combinator ([Kennaway82]) that takes its third argument (V) and directs it towards both E_1 and E_2. If either E_1 or E_2 does not require V then it must reject it (using the K combinator). Turner improved the quality of the code considerably by introducing selective director combinators: that is, combinators that direct the argument only where it is required.

One of the attractions of Turner's combinators is that there are few enough of them to form the machine code of a computer. This has been exploited by Clarke, Gladstone, MacLean and Norman in the SKIM machine [Clarke80]. This is a bitslice uni-processor with a cell-structured memory microprogrammed to execute Turner's combinators. SKIM is programmed in SMALL, a SASL-like applicative language, and compiles the SMALL compiler (itself written in SMALL) in about a quarter of an hour. An improved version is expected to offer comparable performance to a 68000 running conventional languages.

Jones and Muchnik have taken a rather different approach [Jones82]. The compile each combinator into more primitive operations of a stack machine code and then optimise the code produced. This is a hybrid between conventional compilation and combinator reduction. The authors have not yet compared its efficiency to direct combinator interpretation, or to SKIM.

The view of combinators as 'directors' has been taken further by Kennaway and Sleep [Kennaway82]. They represent a director as one of ↑, /, \, and – indicating that an argument is to be directed to both, the left, the right, or neither of the branches of a function application. They use strings of such directors as combinators, indicating the appropriate directors for successive arguments. Director strings are attractive as combinators because they can be represented very concisely, using only two bits per director.

These director strings will be used in Burton and Sleep's ZAPP (Zero Assignment Parallel Processor) [Burton81]. ZAPP consists of a large number of processors, each with considerable local memory. The processors are connected in a cyclic network that allows each processor to see an infinite virtual binary tree of other processors. As in AMPS, each processor maintains a list of pending processes. When a processor becomes lightly loaded it may steal a pending process from one of its neighbours, with the restriction that a stolen process may never be re-stolen. This restriction guarantees that a process never migrates further than one network communication from its origin, and therefore that its result need only be transmitted one network step once it is computed. Since each processor sees an infinite binary tree, work can spread through the network very quickly. ZAPP does not support a global address space: instead all data a process may need is stolen along with the process. Herein lies the weakness of the design, for data can be stolen before it is computed. One unevaluated datum can be stolen many times by many different processors, forcing them all to evaluate it and multiplying the amount of work to be done. Thus the attempt to ensure locality of reference risks enormously duplicated computation. It remains to be seen whether this will occur in practice.

ZAPP also incorporates an elegant device for controlling parallelism automatically. Burton and Sleep observe that breadth-first computation leads to lots of parallelism, but also lots of space utilisation; on the other hand, depth-first computation is very space efficient but gives rise to no parallelism. ZAPP compromises by assigning breadth-first processes a lower priority than depth-first ones, and executing higher priority processes by preference. This leads to breadth-first computation until all processors are in use, followed by depth-first computation, resulting in a space requirement proportional to the number of processors.

9.6. SUPER-COMBINATOR APPROACHES

We have not yet made a realistic implementation of a functional language using super-combinators, and nor, to our knowledge, has anyone else. However, Johnsson has independently developed a very similar technique which he uses in his ML compiler [Johnsson83] (actually he compiles a purely functional dialect of ML with lazy evaluation). Johnsson generates combinators from ML programs by a process he calls "lambda lifting", which is analogous to generating super-combinators using only the methods of sections 3.5 and 4.5. The combinators are then compiled into machine code for the G-machine, an abstract machine with a stack in addition to the graph storage, which performs the associated graph transformation. The G-machine code is translated into VAX machine code and called from a graph reduction interpreter. However, Johnsson goes to a great deal of trouble to optimise the G-machine code so that it avoids constructing pieces of graph which can safely be reduced immediately: for example, the function succ defined by

```
succ n = n + 1
```

is compiled into G-machine code which never constructs the expression n+1, but just computes its value straight away. This, together with other efforts to optimise the code produced, makes Johnsson's lazy ML run as fast as more conventional languages. Johnsson is considering other implementations of the G-machine, including an implementation in hardware and a parallel version.

In addition, Fairbairn will shortly be using super-combinators in an implementation of his language Ponder [Fairbairn82] for the Motorola 68000. Like Johnsson, he will make great efforts to execute each combinator more efficiently than by simple graph transformation.

9.7. DATAFLOW

A very different view of functional language implementation is embodied in dataflow designs. The original idea of dataflow was that the program would be represented by a network of instructions through which data would flow, and out of which answers would emerge. Since many parts of the graph might be active at the same time the approach offers good prospects for parallelism. Dennis et al. designed a machine based on these principles described in [Dennis79], which consisted of eight processors connected by a switching network like the one used in ALICE and DSI. Its disadvantage is inherent in the original conception: since the program is static, it cannot contain invocations of recursive or higher-order functions.

Watson and Gurd proposed a partial solution to this problem in [Watson79], whereby data is "coloured" with the particular function invocation it belongs to. This allows multiple concurrent activations of the same piece of program graph, and so recursion is possible, but this scheme still does not support higher-order functions naturally. Watson and Gurd have built a single processor machine to their design, which achieves high speed by parallelism inside the processor. Extension to a multiprocessor version would present no problems. An important point about their design is that one processor executes several processes simulteneously, and so it is not held up while a communication, such as a memory access, is in progress. This means that their communication network, while it must have high throughput, need not necessarily complete individual transactions quickly.

In [Dennis81] Dennis proposes a new kind of dataflow architecture in which the program graph changes dynamically. It differs from previous dataflow designs in that a dataflow graph represents an object, which reduces to a value, rather than a function through which data must flow. It is able to support higher-order functions and so on, and in fact, it is really a graph reduction architecture of the kind we have been discussing in this thesis. The major difference is that evaluation is data-driven rather than demand-driven, that is, all reducible expressions in the program are evaluated in parallel, whether their results are really needed or not. We suspect that some method of restraining parallelism will prove necessary, whereupon dataflow architectures and other graph reduction architectures will have converged almost completely.

CHAPTER 10

CONCLUSION

Our first conclusion is that super-combinators provide a reasonable and efficient implementation method, superior at least to Turner's combinators. We have demonstrated this experimentally and theoretically, and other work (section 9.6) supports our conclusion.

Of particular interest is the discovery of the "slow-down factor" (chapter 6), in which we refused to believe at first. We originally thought that no slow-down factor was the mark of a good implementation method, and did not believe that Turner's combinators suffered from one until we saw Kennaway's proof. We thought that super-combinators had no slow-down factor until we saw Fischer's example. Only after being wrong twice did we re-examine other approaches and observe the same behaviour in them all. We conclude from this that it is dangerous to design implementation strategies solely to make individual operations fast. When a new strategy is proposed it should be accompanied by a careful analysis of its slow-down behaviour.

We believe we have demonstrated conclusively that functional abstract machines must support parallelism, and that there is strong evidence that this parallelism must be controlled explicitly by the programmer. This can be done adequately using our PAR/SYNCH constructions, which have the important property that they can be added to a working program; they need not be considered during the design. However, a more structured alternative might prove easier to use.

In the more distant future mark-scan garbage collection will become impractical due to the size of virtual memories that will need to be garbage collected. Our final conclusion is that an extension of reference counting can be used instead, at a moderate additional cost.

REFERENCES

[Backus78] J. Backus. "Can programming be liberated from the von Neumann style? A functional style and its algebra of programs". Communications of the ACM. Vol. 21. No. 8

[Berkling] K. J Berkling. "A Complete Abstract Lambda Calculus Machine".

[Bobrow80] D. G. Bobrow. "Managing reentrant structures using reference counts". ACM Toplas Vol 2. No. 3. July.

[Brownbridge83] D. Brownbridge. Private communication.

[Burstall80] R. M Burstall. D. B. MacQueen. D. T. Sanella. "HOPE. an experimental applicative language". Proceedings of the ACM Lisp Conference.

[Burton81] F. W. Burton. M. R. Sleep. "Executing functional programs on a virtual tree of processors". Proceedings of the ACM Conference on Functional Programming Languages and Computer Architectura.

[Burton82] F W. Burton. "A linear space translation of functional programs to Turner combinators".

[Clarke80] T. J. W. Clarke. P. J. S. Gladstone. C. D. MacLean. A. C. Norman. "SKIM – the S. K. I reduction machine". Proceedings of the 1980 Lisp Conference.

[Curry58] H. B. Curry, R. Feys, "Combinatory Logic", North-Holland Publishing Company, Amsterdam

[Darlington81] J. Darlington, M. Reeve, "ALICE - a multi-processor reduction machine for the parallel evaluation of applicative languages", Proceedings of the 1981 ACM Conference on Functional Programming Languages and Computer Architecture.

[Dennis79] J. B. Dennis, "The varieties of data-flow computers", Proceedings of the 1st International Conference on Distributed Computing.

[Dennis81] J. B. Dennis, "Data should not change - a model for a computer system", Symposium on Functional Languages and Computer Architecture.

[Dennis82] J. B. Dennis, Private communication

[Dijkstra79] E. W. Dijkstra, L. Lamport, A. J. Martin, C. S. Scholten, E. F. M. Steffens, "On the fly garbage collection: an exercise in cooperation", Communications of the ACM Vol. 21 No. 11.

[Fairbairn82] J. Fairbairn, "Ponder and its type system", University of Cambridge Computer Laboratory Technical Report No. 31

[Fischer72] M. J. Fischer, "Efficiency of equivalence algorithms", Complexity of Computer Computations, Plenum Press, New York.

[Friedman79] D. P. Friedman, D. S. Wise, "Reference counting can manage the circular invironments of mutual recursion", Information Processing Letters, Vol. 8, No. 1.

[Friedman80] D. P. Friedman, D. S. Wise, "An indeterminate constructor for applicative programming", 7th Annual Symposium on Principles of Programming Languages.

[Gordon79] M. Gordon, R. Milner, C. P. Wadsworth, "Edinburgh LCF", Springer-Verlag Lecture Notes in Computer Science Vol. 78.

[Grit81] D. H. Grit, R. L. Page, "Deleting irrelevant tasks in an expression-oriented multi-processor system", ACM Topics Vol. 3, No. 1.

[Henderson76] P. Henderson, J. H. Morris, "A lazy evaluator", Proceedings of the 3rd Annual SIGACT-SIGPLAN Symposium on Principles of Programming Languages, Atlanta.

[Henderson80] P. Henderson, "Functional programming: application and implementation", Prentice-Hall.

[Henderson83] P. Henderson, G. A. Jones, S. B. Jones, "The lispkit manual", Oxford University Programming Research Group Technical Monograph PRG-32.

[Hudak82] P. Hudak, R. M. Keller, "Garbage collection and task deletion in distributed applicative processing systems", Proceedings of the ACM Symposium on LISP and Functional Programming.

[Hughes82] R. J. M. Hughes, "Super-combinators: a new implementation method for applicative languages", Proceedings of the ACM Symposium on LISP and Functional Programming.

[Hughes82] R. J. M. Hughes. "Graph-reduction with super-combinators". Oxford University Programming Research Group Technical Monograph PRG-28.

[Johnson81] S. D. Johnson, A. T. Kohlstaedt. "DSI program description". Indiana University Computer Science Department Technical Report No. 120

[Johnsson83] T. Johnsson. "The G-machine: an abstract machine for graph reduction". Joint SERC/Chalmers University Declarative Programming Workshop, University College London, May

[Jones82] N. D. Jones, S. S. Muchnik. "A fixed program machine for combinator expression evaluation". ACM Symposium on LISP and Functional Programming

[Karlsson81] K. Karlsson. "An outline of the SKY reduction machine". Symposium on Functional Languages and Computer Architecture

[Keller79] R. M. Keller, G. Lindstrom, S. Patil. "A loosely coupled applicative multi-processing system". Proceedings of the 1979 AFIPS Conference

[Kennaway82] J. R. Kennaway, M. R. Sleep. "Director strings as combinators". University of East Anglia.

[Kennaway82] J. R. Kennaway. "The complexity of a translation of λ-calculus to combinators".

[Kohlstaedt81] A. T. Kohlstaedt. "Daisy 1.0 reference manual". Indiana University Computer Science Department Technical Report No. 119.

[Kowalski79] R. A. Kowalski, "Logic for problem solving", North-Holland.

[Landin64] P. J. Landin, "The mechanical evaluation of expressions", Computer Journal, January.

[Liskov79] B. Liskov, R. Atkinson, T. Bloom, E. Moss, C. Schaffert, B. Scheifler, A. Snyder, "CLU reference manual", Massachussets Institute of Technology Laboratory of Computer Science Technical Report MIT/LCS/TR-225.

[Mago79] G. A. Mago, "A network of microprocessors to execute reduction languages", two parts, International Journal of Computer and Information Sciences, Vol. 8, Nos 5, 6.

[Mago81] G. A. Mago, "Copying operands versus copying results: a solution to the problem of large operands in FFPs", Proceedings of the 1981 ACM Conference on Functional Programming Languages and Computer Architecture

[Peyton-Jones82] S L. Peyton-Jones, "An investigation of the relative efficiencies of combinators and λ-expressions", Proceedings of the ACM Symposium on Lisp and Functional Programming.

[Reynolds70] J. C Reynolds, "GEDANKEN - a simple typeless language based on the principle of completeness and the reference concept", Communications of the ACM, Vol. 13, No. 5, May.

[Steele78] G. L. Steele Jr., G. J. Sussman, "The revised report on SCHEME: a dialect of LISP", Massachussets Institute of Technology Artificial Intelligence Laboratory Memo 452.

[Steele80] G. L. Steele Jr., G. J. Sussman. "Design of a LISP based microprocessor". Communications of the ACM. Vol. 23. No 11

[Sufrin83] B. A Sufrin. "Reading formal specifications". Oxford University Programming Research Group Technical Monograph PRG-24.

[Tarjan72] R. E. Tarjan. "Depth first search and linear graph algorithms". SIAM Journal of Computing Vol. 1. No 2

[Tarjan75] R E Tarjan. "Efficiency of a good but not linear set union algorithm". Journal of the ACM. Vol. 22. No 2. April

[Turner76] D. A Turner. "SASL language manual". St. Andrews University.

[Turner79] D A. Turner. "A new implementation technique for applicative languages". Software. Practice and Experience. Vol. 9.

[Turner79] D. A. Turner. "Another algorithm for bracket abstraction", Journal of Symbolic Logic. Vol. 44. No. 2. June.

[Turner81] D A. Turner. "Recursion equations as a programming language". Newcastle Summer School in Functional Programming.

[Wadler83] P. L Wadler. "Listlessness is better than laziness". Ph.D. Thesis. Carnegie-Mellon University

[Wadler83] P L Wadler. private communication.

[Wadsworth71] C. P. Wadsworth, "Semantics and pragmatics of the λ-calculus", Oxford University D. Phil. thesis.

[Watson79] I. Watson, J. Gurd, "A prototype dataflow computer with token labelling", Proceedings of the AFIPS Conference.

[Wise81] D. S. Wise, "Compact layouts of banyan/FFT networks", VLSI Systems and Computations, Computer Science Press, Rockville, Maryland.

Appendix

A.1 Introduction

These appendiees contain details of the experiments referred to briefly in section 4.6. The experiments were performed during 1981, and were intended to compare the performance of an SKI implementaion with a super-combinator implementation. Since I did not originally intend to include any more than the results of these experiments in my thesis I did not preserve listings of the programs used or samples of their output. Instead, these appendices consist of an explanation of the experiments in sufficient detail to allow the reader to repeat them.

We begin in section A.2 with a brief description of the source language we compiled, and some sample programs. In section A.3 we will describe the compiler used to generate SKI and super-combinator code. Since we cannot reproduce a listing of it, we include in section A.4 a listing of a working compiler written in Prolog, which compiles Lispkit Lisp into super-combinators. In section A.5 we discuss the abstract machine-code interpreter and the statistics we gathered. Finally, in section A.6 we present the experimental results in more detail.

A.2 Nose: the language and examples

The functional language which we compiled in our experiments was Nose (an acronym for NO Side Effects). Nose was designed at Cambridge in 1980 as part of a Diploma in Computer Science [Hughes80]. It draws heavily on the applicative subset of PAL [PAL ref] for its syntax, and so should appear familiar to anyone knowing PAL, ISWIM [Landin66] or SASL [Turner76]. Like PAL, Nose has fixed length tuples rather than lists, and these must be chained together for list processing. Unlike PAL, Nose is purely functional and has a lazy semantics. Nose type checking is performed at run-time.

Rather than give a precise definition of Nose, we will indicate its flavour by giving several example programs. The first of these is a program to compute Ackerman's function. This program was actually used in our experiments.

```
let rec ack m n =
        if m=0 then n+1
        elif n=0 then ack (m-1) 1
        else ack (m-1) (ack m (n-1))
        fi
in ack 2 3
```

This is a curried version of Ackerman's function. When programming in Nose we tended to write non-curried functions instead, using Nose tuples as argument lists. The following program is a non-curried version of Ackerman's function, and was also used as an example in our experiments.

```
let rec ack [m,n] =
        if m=0 then n+1
        elif n=0 then ack [m-1,1]
        else ack [m-1,ack[m,n-1]]
        fi
in ack [2,3]
```

As an example of list processing in Nose, we include the following program for computing primes.

```
show [20,sieve(from 2)]
where rec
    (   from n = [n,from(n+1)]
also    sieve [p,x] = [p,sieve(filter x)]
                         where rec filter [n,x] =
                                 if n%p=0 then filter x
                                 else [n,filter x]
                                 fi
also    rec show [n,l] =
                 if n=0 then "_"
                 else [l1,", ",show[n-1,l2]]
                 fi
    )
```

This program, which uses Eratosthenes' sieve, was adapted from one in

[Turner79a]. It illustrates the use of Nose's indexing operator (!) to access components of a pair. This program was also used in our experiments.

One of our most complex examples was a functional unification algorithm. The program we actually used has been lost, but a similar Nose definition might be:

```
let rec unify [a,b,env] =
#       a and b are expressions to be unified. env is either 'empty' (of type
#       void), or it is a tuple of bindings of type [string,any]*. unify returns
#       a new environment which is an extension of env in which a and b
#       are equal (after substitution of values for variables), or 'empty' if
#       no such environment exists. Variables are strings beginning with "*".
        kind env in
                empty:void -> empty
        |       bindings:[string,any]* ->
                        if a=b then bindings
                        elif isvar a then
                                if bound[a,bindings] then
                                        unify[lookup[a,bindings],b,bindings]
                                else [a,b] pre bindings
                                fi
                        elif isvar b then unify[b,a,bindings]
                        else kind [a,b] in
                                [atup:any*,btup:any*] ->
                                        if length atup=length btup then
                                                uu[1,bindings]
                                            where rec uu[i,env'] =
                                                if i>length atup then env'
                                                else unify[atup!i,btup!i,
                                                        uu[i+1,env']]
                                                fi
.                                       else empty
                                        fi
                            |   other:any -> empty
                            dnik
                        fi
        dnik
where bound [a,env] = (bound' 1
        where rec bound' i =
                if i>length env then false
                else let [a',v'] = env!i in
                        if a=a' then true else bound' (i+1) fi
                fi)
also lookup [a,env] = (lookup' 1
        where rec lookup' i =
                let [a',v'] = env!i in
                if a=a' then v' else lookup' (i+1) fi
also isvar a =
        kind a in
            s:string -> stem s = "*"
        |   other:any -> false
        dnik
```

This example demonstrates Nose "kind-expressions". They are the only type-testing mechanism Nose provides.

These examples should be enough to satisfy the reader's curiosity. We do not claim that Nose is a particularly good language. Several aspects of its design appear wrong in retrospect - for example, the use of fixed length tuples instead of lists, and the unwieldy kind-expressions. It was only an experimental tool, and it served its purpose.

A.3 The Nose compiler

The Nose compiler was originally written at Cambridge to investigate optimisation methods for lazy languages [Hughes80]. It was written in GEDANKEN and run on the CPU GEDANKEN system, developed at Cambridge between 1976 and 1980. It was ported to Oxford by writing an abstract machine-code interpreter for GEDANKEN code.

The compiler was extended to compile to either SKI combinators or super-combinators. First a common pass transformed declarations into lambda-expressions and syntactic constructs into calls of standard functions (see section 3.2). The output of this pass was an expression in the pure lambda-calculus with constants. Since everything up to this stage was common to the SKI and super-combinator compilers, no extraneous factors influenced our experiments.

The lambda-expressions were translated into SKI combinators using the methods of [Turner79a] and [Turner79b]. That is, Turner's later "optimising" combinators (S', B' and C') were used. The lambda-expressions were translated into super-combinators using the imperative algorithm of section 5.3. The super-combinators were then compiled into BCPL code that, when executed, built the result of applying the combinator. The BCPL code was called by the combinator interpreter at run-time. An earlier version interpreted super-combinator definitions: compiling to BCPL improved speed by 15%. The improvements of sections 4.4 and 4.5 were not used, and in some cases this has biased our results against super-combinators. This is most noticeable in the case of Ackerman's function (see section A.6).

A deficiency of our compiler was that Nose tuples were compiled as lists. Since list processing programs in Nose chain tuples together this introduced an unnecessary overhead into the code produced by both compilers, which will have distorted our results somewhat. Differences in efficiency which we have observed would be more pronounced of this had been corrected.

Secondly, our compiler was very slow due to the use of a BCPL interpreter for GEDANKEN abstract machine-code. This, coupled with the small amount of store available (about 25K words) limited the size of example we could try severely. Our results may therefore be distorted by "end-effects". We have interpreted them by looking for trends as program size increases.

A.4 A Lispkit compiler in Prolog

The Nose compiler described in section A.3 was far too large to include as an appendix. So that the reader my run a super-combinator compiler if he so wishes, we have included a version written in Prolog. This compiler is the model for section 5.5. It compiles Lispkit Lisp rather than Nose, but is otherwise very similar.

COMPILE.PRO

```
/*      This file contains the steering program of the Lispkit compiler. Consult it, and then
        invoke the compiler by, for example, compile('frolo.lso','frolo.sup').
*/

?- consult(lists). /* standard list-processing functions */
?- consult(lispkit).
?- consult(sup).
?- consult(optimise).
?- consult(writecode).
?- consult(env).

compile(From,To) :-
        write('LISP Compiler to Super-combinators'), nl,
        write(From), write(' ----> '), write(To), nl,
        (docompile(From,To), write('Compilation succeeds'), nl;
         write('Compilation failed'), nl).
docompile(From,To) :-
        see(From), read(Prog), seen,
        write('Syntax Analysis complete'), nl, !,
        preproc(Prog,X), lispkit(X,Lam),
        write('Converted to lambda calculus'), nl, !,
        sup(nile,Lam,expr(0,[],Sup,_),[]),
        write('Compiled to super-combinators'), nl, !,
        optimise(Sup,Opt),
        write('Optimisation complete'), nl, !,
        tell(To), writecode(Opt), nl, told, !
```

LISPKIT.PRO

```
/*      This file defines the predicate lispkit(A,B), which converts a lispkit program
        into lambda-calculus ready for compilation.
*/

lispkit(Name,quote(Name)) :- standard(Name).
lispkit(Int,quote(constant(Int))) :- integer(Int).
lispkit(Name,Name) :- atom(Name).
lispkit([quote,E],quote(constant(E))).
lispkit([lambda,[],Body],Body1) :- lispkit(Body,Body1).
lispkit([lambda,Name.Names,Body],lambda(Name,Body1)) :-
        lispkit([lambda,Names,Body],Body1).
lispkit([let,Body|Defs],E) :-
        decompose(Defs,Names,Vals),
        lispkit([lambda,Names,Body],E1),
        applist(E1,Vals,E).
lispkit([letrec,Body|Defs],apply(Body2,apply(quote(fix),Vals2))) :-
        decompose(Defs,Names,Vals),
        lispkit(Body,Body1),
        lamlist(Names,Body1,Body2),
        makelist(Vals,Vals1),
        lamlist(Names,Vals1,Vals2).
lispkit([F],F1) :- lispkit(F,F1).
lispkit([F,A],apply(F1,A1)) :- lispkit(F,F1), lispkit(A,A1).
lispkit([F,A|R],E) :- R\=[], lispkit([[F,A]|R],E).

applist(E,[],E).
applist(E,V.Vs,E1) :- lispkit(V,V1), applist(apply(E,V1),Vs,E1).

lamlist([],F,apply(quote(k),F)).
lamlist(A.B,F,apply(quote(unpack),lambda(A,F1))) :- lamlist(B,F,F1).

makelist([],quote(nil)).
makelist(A.B,apply(apply(quote(cons),A1,B1)) :-
        makelist(B,B1), lispkit(A,A1).

decompose([],[],[]).
decompose([A.B|C],[A|N],[B|V]) :- decompose(C,N,V).

standard(add).
standard(sub).
standard(div).
standard(mul).
standard(rem).
standard(eq).
standard(less).
standard(greater).
standard(cons).
standard(car).
standard(cdr).
standard(if).
standard(which).

preproc(A,A) :- atomic(A).
preproc(A.B,A1.B1) :- preproc(A,A1), preproc(B,B1).
preproc(A,C) :- A=..B, preproc(B,C).
```

SUP.PRO

```
/*      This is a compiler to super-combinators written in Prolog. Input syntax:
                atom     quote(const)      apply(fn,arg)    lambda(id,e)
        Output syntax:
                super(nargs,body)          arg(n)
*/

/*      sup(Env,In,Out,Names) takes an environment and an input expression, and
        computes an output expression of the form expr(level,mfes,object,source).
        Names is isomorphic to mfes.
*/

sup(Env,Id,expr(L,[],arg(N),Id),[]) :- atom(Id), lookup(Env,Id,L,N), !.
sup(_,quote(Const),expr(0,[],quote(Const),quote(Const)),[]) :- !.
sup(Env,apply(Fn,Arg),Expr,Names) :-
        sup(Env,Fn,FnExp,FnNames), sup(Env,Arg,ArgExp,ArgNames),
        supapp(FnExp,ArgExp,FnNames,ArgNames,Exp,Names), !.
sup(Env,lambda(Id,Exp),Expr,Names) :-
        bind(Env,Id,Nargs,Env1),
        sup(Env1,Exp,EBody0,NBody0),
        lookup(Env1,Id,Lev,Nargs),
        lambody(Lev,EBody0,NBody0,EBody,NBody),
        suplam(lambda(Id,Exp),EBody,NBody,Expr,Names,Nargs), !.

supapp(expr(FL,FM,FE,FS),expr(AL,AM,AE,AS),FNms,ANms,
                expr(L,M,apply(FE,AE),apply(FS,AS)),Nms) :-
        (FL=0;AL=0;FL=AL), max(FL,AL,L),
        append(FM,AM,M), append(FNms,ANms,Nms).
supapp(expr(FL,FM,FE,FS),expr(AL,AM,AE,AS),FNms,ANms,
                expr(AL,expr(FL,FM,FE,FS).AM,apply(arg(NN),AE),apply(FS,AS)),
                name(NN,FNms).ANms) :-
        FL<AL, FL\=0.
supapp(expr(FL,FM,FE,FS),expr(AL,AM,AE,AS),FNms,ANms,
                expr(FL,expr(AL,AM,AE,AS).FM,apply(FE,arg(NN)),apply(FS,AS)),
                name(NN,ANms).FNms) :-
        FL>AL, AL\=0.

max(A,B,B) :- A<B.
max(A,B,A) :- A>=B.

lambody(Lev,expr(L,M,E,S),N,expr(L,M,E,S),N) :- Lev=L; L=0.
lambody(Lev,expr(L,M,E,S),N,expr(Lev,[expr(L,M,E,S)],arg(N1),S),[name(N1,N)]) :-
        Lev\=L, L\=0.

suplam(Exp,expr(BL,BM,BE,BS),NBody,Expr,Names,Nargs) :-
        sortmfes(BM,NBody,BM1,NBody1),
        optmfes(BM1,NBody1,BM2,NBody2),
        mkap(expr(0,[],super(Nargs,BE),Exp),BM2,NBody2,Expr,Names,Nargs).

optmfes([],[],[],[]).
optmfes(M.Ms,N.Ns,M.Ms1,N.Ns1) :- not(member(M,Ms)), optmfes(Ms,Ns,Ms1,Ns1).
optmfes(M.Ms,N.Ns,Ms1,Ns1) :- element(Ms,I,M), element(Ns,I,N), optmfes(Ms,Ns,Ms1,Ns1).
```

```
sortmfes([],[],[],[]).
sortmfes([expr(L,M,E,S)|Ms],N,Ms1,Ns1) :-
        splitmfes(L,[expr(L,M,E,S)|Ms],N,SmM,SmN,EqM,EqN,BiM,BiN),
        sortmfes(SmM,SmN,SmM1,SmN1), sortmfes(BiM,BiN,BiM1,BiN1),
        append(BiM1,EqM,BiEqM), append(BiN1,EqN,BiEqN),
        append(BiEqM,SmM1,Ms1), append(BiEqN,SmN1,Ns1).

splitmfes(L,[],[],[],[],[],[],[],[]).
splitmfes(L0,expr(L,M,E,S).Ms,N.Ns,expr(L,M,E,S).SmM,N.SmN,EqM,EqN,BiM,BiN) :-
        L<L0, splitmfes(L0,Ms,Ns,SmM,SmN,EqM,EqN,BiM,BiN).
splitmfes(L0,expr(L0,M,E,S).Ms,N.Ns,SmM,SmN,expr(L0,M,E,S).EqM,N.EqN,BiM,BiN) :-
        splitmfes(L0,Ms,Ns,SmM,SmN,EqM,EqN,BiM,BiN).
splitmfes(L0,expr(L,M,E,S).Ms,N.Ns,SmM,SmN,EqM,EqN,expr(L,M,E,S).BiM,N.BiN) :-
        L>L0, splitmfes(L0,Ms,Ns,SmM,SmN,EqM,EqN,BiM,BiN).

mkap(Comb,[],[],Comb,[],1).
mkap(Comb,Arg.M,name(Nargs1,ArgNames).N,Expr,Names,Nargs) :-
        mkap(Comb,M,N,Expr1,Names1,Nargs1),
        supapp(Expr1,Arg,Names1,ArgNames,Expr,Names),
        Nargs is Nargs1+1.
```

OPTIMISE.PRO

```
optimise(arg(X),arg(X)).
optimise(quote(X),quote(X)).
optimise(apply(F,A),B) :- optimise(F,F1), optimise(A,A1), optapp(F1,A1,B).
optimise(super(N,E),S) :- optimise(E,E1), optsup(N,E1,S).

optapp(super(2,superapply(arg(1),arg(2))),A,A).
optapp(F,A,apply(F,A)) :- constant(F), constant(A).
optapp(F,A,superapply(F,A)).

optsup(1,superapply(F,arg(1)),F) :- constant(F).
optsup(N,E,super(N,E)).

constant(quote(X)).
constant(apply(F,A)).
constant(super(N,E)).
```

WRITECODE.PRO

```
writecode(quote(constant(C))) :- writeconstant(C).
writecode(quote(nil)) :- write('[]').
writecode(quote(Atom)) :- write(Atom).
writecode(arg(N)) :- write('/'), write(N).
writecode(apply(apply(quote(cons),A),B)) :-
        write('['), writecode(A), write('.'), writecode(B), write(']').
writecode(apply(F,A)) :- write('('), writecode(F), write(' '), writecode(A), write(')').
writecode(superapply(apply(quote(cons),A),B)) :-
        write('<'), writecode(A), write('.'), writecode(B), write('>').
writecode(superapply(superapply(quote(cons),A),B)) :-
        writecode(superapply(apply(quote(cons),A),B)).
writecode(superapply(F,A)) :- write('('), writecode(F), write(' '), writecode(A), write(')').
writecode(super(N,E)) :- write('\'), write(N), write(' '), writecode(E).

writeconstant([]) :- write('[]').
writeconstant(A.B) :- write('['), writeconstant(A), write('.'), writeconstant(B), write(']').
writeconstant(Atom) :- atom(Atom), write('"'), write(Atom), write('"').
writeconstant(Int) :- integer(Int), write(Int).
```

ENV.PRO

```
bind(nile,Id,N,env(Id,1,N,nile)).
bind(Env,Id,N,env(Id,L1,N,Env)) :- Env=env(_,L,_,_), L1 is L+1.

lookup(Env,Id,Lev,N) :- includeonly(Env,Id,env(Id,Lev,N,_)).

includeonly(nile,Id,nile).
includeonly(env(Id,Lev,N,Env),Id,env(Id,Lev,N,Env1)) :- includeonly(Env,Id,Env1).
includeonly(env(Id0,_,_,Env),Id,Env1) :- Id0\=Id, includeonly(Env,Id,Env1).
```

A.5 The interpreter

In order to ensure that our measurements of the efficiency of SKI combinators and super-combinators were comparable, we used the same interpreter in each case. Our interpreter was written in BCPL, and was very similar to the one described in [Turner79a]. The program being executed was represented in memory as a graph of application cells. The interpreter reduced an expression by, first of all, following function pointers until it reached a non-application. This would be the function to be applied. The nodes passed were placed on a "left-ancestor stack", from which the arguments of the function could be found. The result of the function application was computed, and the original expression over-written with an "indirection node" referring to this result. These indirection nodes were removed by the garbage collector. This process continued until either an atomic result was computed, or a function was found without enough arguments. At this point, the original expression was fully reduced.

The interpreter contained a number of hand-coded functions for performing basic operations, and hand-coded definitions of all the SKI combinators. When used to execute super-combinator code, the compiled super-combinators were loaded along with the interpreter and called directly by it.

The interpreter made a number of measurements during execution. These were:

> The number of reductions performed. We expect the super-combinator interpreter to do significantly fewer reductions since each super-combinator corresponds to several SKI combinators. The ratio of SKI reductions to super-combinator reductions is an indication of the number of SKI combinators that each super-combinator corresponds to.
>
> The total number of cells claimed. This measure was used by Turner in his comparison of SKI combinators with the SECD machine. It is probably the best machine-independent measurement of efficiency.
>
> The number of garbage collections and the maximum number of cells in use simultaneously, measured at each garbage collection. This figure was intended to reveal the amount of store actually required by each program. Unfortunately, our compiler restricted us to such small programs that garbage collections at run-time were rare, and no meaningful figures were obtained.
>
> The run-time in seconds.

The results of these measurements, and other measurements performed by the compiler, are summarised in the next section.

A.6 Experimental results

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

Program	Size	Purpose
1	26	call "twice" [Turner79a]
2	36	Ackerman's function (curried)
3	49	towers of hanoi
4	51	Ackerman's function (non-curried)
5	75	factorial
6	93	append
7	106	20 primes
8	115	eratosthenes' sieve
9	307	unification algorithm
10	317	e to 20 decimal places

Table II. Compile-time in seconds.

Program	Size	SKI	SC	%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	SKI	SC	%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	SKI	SC	%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	SKI	SC	%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	SKI	SC	%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

A.7 References

[Hughes80] R.J.M.Hughes, The Design and Implementation of an Applicative Language, Cambridge University Diploma Dissertation.

[Landin66] P.J.Landin, The Next 700 Programming Languages, Communications of the ACM, 9(3), 157-164, March 1966.

[Turner76] D.A.Turner, SASL Language Manual, St.Andrew's University.

[Turner79a] D.A.Turner, A New Implementation Technique for Applicative Languages, Software: Practice and Experience, Vol. 9.

[Turner79b] D.A.Turner, Another Algorithm for Bracket Abstraction, Journal of Symbolic Logic, 44(2), June 1979.

[PAL ref]