

The Implementation of Functional Languages on an Object-oriented Architecture

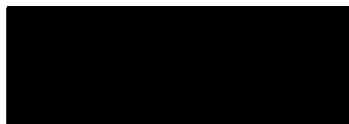
Mohammed Yousuf Khan

A Thesis submitted in partial fulfilment of the
requirements of Dundee Institute of Technology
for the Degree of Doctor of Philosophy

March 31, 1993

I certify that this thesis is the true and accurate version of the
thesis approved by the examiners.

Signed: .



..

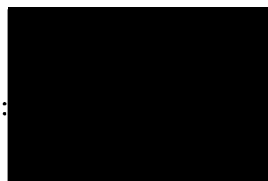
date: *12 April 1993*

(Director of Studies)

Declaration

I declare that while registered as a candidate for the degree for which this thesis is presented, I have not been a candidate for any other award. I further declare that except where stated the work contained in this thesis is original and was performed by the author.

Signed:

A solid black rectangular box used to redact the signature of the author.

(M. Y. Khan)

Acknowledgements

The author wishes to thank his supervisors, Dr. W. B. Samson, Mr. A. C. Milne, and Mr. L. D. Natanson, for their invaluable support, encouragement and guidance given at all times throughout the duration of this project. A special thanks for their keen interest in the development of the project, without which this research would not be possible.

The author is greatly indebted to his external advisors Mr. G. Michaelson and Professor D. M. Harland for their valuable advice received on various aspects of the investigation.

The author would also like to thanks the reviewers, who suggested many improvements to this thesis, including the addition of several interesting experiments, with special thanks to P. I. Dugard for her valuable advice on statistical analysis.

Thanks to the technical staff, in particular Mr J. Duncan for his assistance in the Laboratory.

Abstract

This thesis concerns the investigation of different implementation strategies for functional languages on a novel platform, the Rekursiv, that provides hardware support for an object-oriented language (Lingo) which was used as the implementation language for this work.

The objective is to evaluate the performance of two different implementation techniques - fixed combinators and supercombinators - for functional languages by interpreting the functional program in two object-oriented styles with different representations for combinators on two different environments:

- A purpose-built object-oriented machine; Harland's Rekursiv.
- A RISC machine; an IBM RS6000 running a Smalltalk-80 interpreter.

Interpreters are implemented in Smalltalk (for the RS6000) and Lingo (for the Rekursiv) for an applicative language Alcal (a λ -calculus language). The interpreters are then used to generate a combinator graph using the fixed SKI set of combinators proposed by Turner, and also Hughes-style program specific supercombinators (λ -lifting) which are not limited to a fixed set of primitive combinators.

A number of experiments are conducted, based on the implementation of the lazy functional language, Alcal, on the RS6000 in Smalltalk-80 and on the Rekursiv architecture in Lingo. Lingo and Smalltalk allow the creation of objects that contain executable statements, so an object-oriented 'active graph' implementation is proposed.

Using these prototypes, the performance of the Rekursiv implementation against the RISC implementation has been evaluated. This includes the proportion of time spent on overhead activities and a statistical performance analysis of the benchmark results to analyse some of the claims made by the designer of the Rekursiv architecture.

Relative to a baseline benchmark written in C, the implementations on the Rekursiv are found to be several times faster than those on the RISC machine. An analysis of variance demonstrates a more subtle interplay between benchmark programs, implementation techniques, representation of combinators and hardware platform.

Contents

1	Background	1 - 1
1.1	History	1 - 1
1.2	Functional programming languages.	1 - 5
1.2.0.1	Mathematical function definitions	1 - 5
1.2.0.2	Referential Transparency	1 - 7
1.2.0.3	High productivity	1 - 7
1.2.1	λ -calculus and functional languages.	1 - 7
1.2.2	The von Neumann Bottleneck.	1 - 8
1.2.3	Concurrency	1 - 8
1.3	Novel machine structures	1 - 9
1.3.1	Architectural aim	1 - 9
1.3.1.1	Sound theoretical basis	1 - 9
1.3.1.2	Efficient resource allocation	1 - 10
1.3.1.3	Hardware support for applicative languages	1 - 11

1.3.1.4	Variable length objects	1 - 11
1.3.1.5	Type checking	1 - 12
1.4	The Object-oriented paradigm and machine architecture.	1 - 12
1.4.0.1	Data abstraction	1 - 13
1.4.0.2	Communications	1 - 13
1.4.0.3	Methods	1 - 14
1.4.0.4	Encapsulation	1 - 15
1.4.0.5	Classes and Instances	1 - 15
1.4.0.6	Inheritance	1 - 15
1.4.0.7	Polymorphism	1 - 16
1.4.1	The Lingo Language	1 - 16
1.4.2	The Rekursiv	1 - 17
1.4.2.1	Memory management	1 - 22
1.4.2.2	Objects on the Rekursiv	1 - 22
1.4.2.3	The virtual memory	1 - 23
1.5	Some of Harland's claims	1 - 24
1.5.1	Features to be investigated	1 - 28
1.6	Aim.	1 - 28
1.7	Summary	1 - 29

2	The implementation environment	2 - 1
2.1	The λ -calculus: Introduction and its syntax	2 - 1
2.1.1	The evaluation of λ -expressions	2 - 3
2.1.1.1	Conversion rules	2 - 4
2.1.2	Different evaluation paths	2 - 5
2.1.2.1	Lazy evaluation	2 - 7
2.2	Implementation techniques	2 - 9
2.2.1	Graph reduction	2 - 10
2.2.1.1	Use of combinators in graph reduction	2 - 12
2.2.1.2	Active graph implementation	2 - 13
2.2.2	Other Techniques	2 - 14
2.2.2.1	SECD machine	2 - 14
2.2.3	Data flow	2 - 16
2.3	Summary	2 - 16
3	Combinator Reduction	3 - 1
3.1	Graphical representation of λ -expressions	3 - 1
3.1.1	String reduction	3 - 3
3.1.2	Graph reduction on Rekursiv	3 - 4
3.2	Combinator reduction scheme	3 - 5

3.3	λ -calculus Vs combinator reduction	3 - 5
3.4	Turner's reduction scheme	3 - 6
3.4.1	Turner's optimization	3 - 9
3.4.2	Representation of recursion	3 - 11
3.5	Supercombinators	3 - 12
3.5.1	Implementation	3 - 14
3.5.2	Parameter ordering and Redundant parameters	3 - 16
3.5.2.1	Redundant parameter	3 - 16
3.5.2.2	Parameter ordering	3 - 17
3.5.3	Lifting with recursion	3 - 18
3.5.4	Conditional handling and functional application	3 - 19
3.5.5	Identifying lambda-abstractions	3 - 19
3.5.6	Identifying free variables	3 - 20
3.5.7	The Evaluator	3 - 20
3.5.8	Advantages over SKI-combinators	3 - 22
4	An Implementation of Combinator Reduction	4 - 1
4.1	The language; syntax and example	4 - 1
4.1.1	Naming the function	4 - 3
4.1.2	Use of brackets	4 - 4

4.1.3	Microsyntax	4 - 4
4.1.4	Distinguishing identifier and variable	4 - 5
4.1.5	Syntax error	4 - 6
4.2	Representation of function definitions	4 - 7
4.2.1	Occurences of formal parameters	4 - 7
4.2.2	Multi parameters	4 - 7
4.2.3	Higher-order functions	4 - 8
4.2.4	Booleans and the conditions	4 - 8
4.2.5	List notations	4 - 9
4.2.5.1	List operators	4 - 10
4.2.5.2	Standard list handling functions	4 - 10
4.2.6	Recursion handling	4 - 11
4.2.7	Intermediate code in prefix form	4 - 12
4.3	Abstract representation	4 - 13
4.3.1	Code optimization	4 - 17
4.4	An active graph implementation	4 - 20
4.5	Class hierarchy and types of node	4 - 23
4.6	Evaluation strategy	4 - 24

5 Performance Evaluation

5 - 1

5.1	Description of experiments	5 - 1
5.1.1	Code generation	5 - 3
5.1.2	Code evaluation	5 - 4
5.2	Overheads	5 - 6
5.3	Experiments	5 - 8
5.3.1	λ -lifting Vs SKI-implementation	5 - 9
5.3.2	Class representation Vs Instance representation	5 - 17
5.3.3	How does Rekursiv differ from RISC?	5 - 22
5.3.3.1	Improvement with the Rekursiv/RISC architectures over different implementations	5 - 23
5.3.3.2	Improvement with the Rekursiv/RISC architectures over different (object-oriented) styles	5 - 24
5.3.3.3	Generating a performance base line for the Rekursiv and the RISC	5 - 24
5.3.4	Regression analysis	5 - 26
5.3.5	The effect of garbage collection	5 - 28
5.3.6	Analysis of variance	5 - 33
5.3.6.1	Factors in model	5 - 33
6	Conclusions	6 - 1
6.1	Future work	6 - 6

6.2	Contribution of this thesis	6 - 7
-----	---------------------------------------	-------

References		Bib - 1
------------	--	---------

Appendices		A - 1
------------	--	-------

Appendix - A	BNF syntax for Alcal	A - 1
Appendix - B	Lingo code for interpreter	B - 1
Appendix - C	Some representative functions (Benchmarks)	C - 1
Appendix - D	Experimental results	D - 1

Chapter 1

Background

This chapter gives a general overview of functional programming languages; implementation aspects, machine structure, and particularly the suitability of the object-oriented paradigm and its associated machine structure for the implementation of functional languages. The final part of this chapter describes the combination of the object-oriented paradigm and functional programming languages and the order in which the work is described in the succeeding chapters.

1.1 History

Programming languages can broadly be classified into three groups, imperative languages, functional languages and logic programming languages.

Most of the commonly used programming languages like FORTRAN, COBOL, BASIC, PASCAL, ALGOL60, SIMULA67 are categorised as imperative languages. They are all built upon a similar computational substrate. The fundamental underlying basis of these imperative languages is on assignment and sequence operations. That is, a program is a sequence of commands or instructions which is to be exe-

cuted. In this respect they resemble the underlying machine on which they run.

Imperative languages (also called ‘conventional’ programming languages) can be implemented efficiently on von Neumann architectures, where a central processor operates on values held in storage locations. This means that the aim of an imperative program is to effect changes in the store of a computer by suitable sequences of assignment statements [33].

In the second group, functional languages, the fundamental operation is function application. Function application means the application of a function to its argument. For this reason they are also known as applicative languages. Sometimes (in the literature) they are also called data flow languages or reduction languages.

The third group, logic programming languages, are beyond the scope of this thesis.

Functional programs contain no assignable program state; in fact they consist of definitions of functions and other objects.

The values of expressions in functional programs are independent of computational history, thereby eliminating a major source of bugs. This referential transparency also makes the order of execution irrelevant and makes functional programs more tractable mathematically than their imperative counterparts [27]. Consider the order of assignments in the following example:

```
k := 0; f := 1;
while k < n do begin k:= k+1; f:= k*f; end
```

If one changed the order of the assignments in the while loop, a completely different program would emerge [67].

To produce high quality software at reasonable cost, Backus [4] has argued that conventional languages are unnecessarily difficult to program in, and that many of

the difficulties stem from the ‘von Neumann’ orientation of the languages concerned [24].

The development of the first functional language LISP started in the late 1950’s by J. McCarthy, which in turn grew out of a theory of functions called the λ -calculus, which was also influenced by recursive function theory (developed by Kleene in 1936 [79]). It was the logician Alonzo Church during the 1930’s who developed λ -calculus which had a great influence on functional languages [14]. LISP is not a pure functional language, it contains imperative as well as functional elements. At present, LISP is mainly used by the artificial intelligence community.

Since then many popular dialects have been developed. For example

- ISWIM: P. Landin, in mid 1960’s, which has been a base for many functional languages [80].
- POP-2: (partially functional) Popplestone and Burstall, 1971, as an updated LISP, which led to POP11 and to Prolog (Edinburgh university) [78].
- SASL: Turner, D. A. in 1976 (St. Andrews University) [82].
- ML: Milner (1978), Edinburgh University, ML is now used as a general purpose functional language like LISP. It has imperative extensions [81].
- Hope: by Burstall (1980) university of Edinburgh, and was designed to be used as the programming language for the ALICE parallel computer [11].
- Miranda: Turner, D. A. in 1985 [66].
- Scheme: developed by Abelson and Sussman, 1986, Scheme still retains some notion of assignments [77].
- Haskell: April 1990, Dept. of Computer Science, Glasgow University: Hudak P, Wadler P L, Arvind, Boutel B, Fairbairn J, Fasel J, Hammond K, Hughes

J, Johnsson T, Kieburtz R, Nikhil R S, Jones S L P, Reeve M, Wise D and Young J [44].

The potential to solve a range of problems is determined by the kind of tools available to the programmer. The way that a problem is addressed is governed by the expressive power of the programming languages to be used.

Comparing functional languages with conventional languages such as Fortran or Pascal or with other “non-conventional” languages such as Smalltalk, most of the high productivity attributed to functional programming is not due to referential transparency, but rather to other properties, such as abstraction, extensibility, higher order functions and heap-allocated memory. On the other hand the context-sensitive aspects of von Neumann languages, particularly assignment statements, require great care and are a major source of error in the implementation of concurrent algorithms in a conventional development environment. von Neumann computers are limited in achieving very high speed execution rates by their intrinsic word-at-a-time processing mechanism.

Most functional languages have their intellectual roots in the λ -calculus. So, λ -calculus is frequently used as an intermediate language in the translation of functional programs [67]. The reason for this is to allow easy comparison of function-evaluation techniques. It is also sufficiently expressive to allow translation of any high-level functional language into it. In principle any function that can be defined in a functional language can be implemented in the λ -calculus. Since Landin (1966) the λ -calculus has been employed as a basis for the design and implementation of functional languages. Languages like **ML**, **Miranda**, **Hope** and **Haskell** have all been inspired by λ -calculus [67].

Growing interest in functional languages represents a radical departure from the concepts prevalent in conventional imperative languages such as Pascal or Ada. Such imperative languages are intimately tied to the Von Neumann model of computation

whereas the origin of functional languages lies in mathematical formalisms developed independently of the construction of computing machinery. One consequence of this is that run-time efficiency of functional programs is poor, compared with imperative programs, where a von Neumann machine is used [15].

It is claimed that by using new types of computers with many processing units, functional programs can be executed faster than conventional programs [72]. For more information, references [30, 43, 68], particularly [68], provide a detailed survey of functional languages and architectures.

1.2 Functional programming languages.

Functional programming languages, like Hope, ML and Miranda, offer an alternative to conventional imperative ones [41]. The most commonly cited advantages of functional languages are:

- mathematical function definitions
- referential transparency
- high productivity

These are discussed below:

1.2.0.1 Mathematical function definitions

Functional programs are built from ‘pure’ functions. One of the fundamental characteristics of mathematical functions is that the evaluation order of their mapping expression is controlled by recursion and conditional expressions, rather than by the sequencing and iterative repetition that is common in imperative programming

languages. Their evaluation cannot alter the environment of the computation ie it is side-effect free. For example consider the following referentially opaque pascal program:

```

program opaque (input, output);
var flag:boolean;
function f(n:integer) : integer;
begin
    if flag then f := n  else f := 2*n
    flag := not flag;
end;

begin (* main *)
flag := True;
writeln ( f(1)+f(2) );
writeln ( f(2)+f(1) );
end.

ie ( f(1)+f(2) ) <> ( f(2)+f(1) )
    1 + 4      <>    2 + 2

```

The expression value depends on the order of evaluation of the two operands.

Computational history is an important aspect of von Neumann languages. Assignment statements, in particular, require great care and are a major source of error in the implementation of algorithms in a conventional development environment.

It would not be possible to write such misleading code in a functional language.

1.2.0.2 Referential Transparency

Referential transparency guarantees that the value of an expression remains the same wherever it occurs within a fixed context in a program. Evaluation of an expression simply changes the form of the expression but never its value. This referential transparency also makes the order of execution irrelevant and makes functional programs more tractable mathematically than their imperative counterparts [27].

Referential transparency has other implications for functional languages. Since the unknown variables in any expressions are simply unevaluated functions calls, which become known as the function code is executed, one effect is to blur the distinction between functions(code) and variables(data). Secondly, static data structures like arrays necessarily have computational histories and so must be replaced in functional languages by dynamic data structures where memory for an item is allocated only when that item comes into existence.

1.2.0.3 High productivity

Most of the high productivity attributed to functional programming is not due to referential transparency, but rather other properties such as abstraction, extensibility, higher order functions, and heap allocated memory [68]. Jones [46] has found that LISP code is, on average, approximately twice as compact as the equivalent C code. If productivities in terms of lines of code written per staff day are the same for both languages, then LISP should demonstrate about twice the productivity of C.

1.2.1 λ -calculus and functional languages.

The functional languages grew out of the language LISP which in turn grew out of a theory of functions called the λ -calculus [33]. Many of the theoretical foundations

of functional languages are based on λ -calculus [22].

Functional languages may be viewed as a user interface to the λ -calculus. Functional programs are built from ‘pure’ functions, ie functions in the mathematical sense, and so the λ -notation is particularly suitable for describing function manipulations formally and even as an intermediate code into which a source program may be translated [22].

An implementation of the λ -calculus must operate on an expression by applying reduction rules to it until no more are applicable. The choice of the order in which reduction rules are applied is a matter of implementation strategy [6].

1.2.2 The von Neumann Bottleneck.

Conventional machines founded on von Neumann principles imply strict sequentiality of operation. A major component of von Neumann computers is a connecting ‘tube’ that can transmit a single word between the CPU and the store. This is often called the von Neumann bottleneck. Programming is thus basically the scheduling through the bottleneck of an enormous traffic of information, much of which is not actually significant data, but addresses used to locate data.

1.2.3 Concurrency

Concurrent execution of functional programs can be effected by first translating them into graphs. These graphs can then be executed through a graph reduction process, which can be done with a great deal of concurrency that was not explicitly specified by the programmer.

In conventional programs it is often not possible to execute a pair of commands in either order because of assignments which alter the environment in which commands

are executed. There is, therefore, little concurrency which can be automatically exploited in a sequence of commands.

The heavy reliance of imperative languages on the underlying architecture is an unnecessary restriction on the process of software development [33].

1.3 Novel machine structures

Quite apart from the issues discussed above, another reason for an interest in functional languages is the suitability of novel machine structures for their implementation.

The question of exactly what sort of machine should be aimed for, needs careful consideration.

One reason for considering new types of architecture is that the run-time efficiency of functional programs is poor, compared with imperative programs, where a von Neumann machine is used.

1.3.1 Architectural aim

An ideal system should have the following basic properties:-

1.3.1.1 Sound theoretical basis

The architecture should be well oriented towards functional languages and furthermore have a sound theoretical basis [51]. The language forms the starting point for the design of the architecture, rather than hardware considerations which take little account of programmability [24].

The declarative languages (λ -based languages and logic based languages) have been considered suitable for novel architectures. λ -based languages do not require backtracking facilities, and are therefore easier to support [24].

Programming in a functional language is much closer to writing a set of mathematical equations than conventional programming. We should make sure that these equations have a workable procedural reading and also have a machine oriented interpretation. This makes it possible to write equations involving infinite objects. This means that if the equations are right, a wrong answer will never be produced, although termination may be affected.

This new approach, however, adds new problems; for example, updating a single element of a huge data structure requires in principle a complete copy of the whole object, and so an appropriate evaluation technique is needed to deal with this.

1.3.1.2 Efficient resource allocation

Functional programming languages offer a high level of abstraction, and place heavy demands on the compile time and run-time strategies by which the machine allocates its resources.

The ideal machine should provide a good automatic resource-allocation strategy. The garbage collection routine is responsible for the re-cycling of memory space which contains information no longer needed for computation. This means that it should make a minimal demand on processor time and minimal access to the main memory. Furthermore the garbage collector should not lock up the processor for significant periods of time. For an implementation which uses a contiguous block of memory a good remedy is a compacting garbage collector. For instance to accommodate an object of a given cell size, the memory heap is scanned for the required space. If it is unavailable the garbage collector may be activated. This situation can be repeated several times during the execution process. With a compacting garbage

collector we can keep a pointer to the beginning of the free-memory block in the heap. If a new cell is needed we can allocate this information in the free-memory block, or in the case of insufficient space garbage collect and compact all the information into another contiguous block of memory [51].

1.3.1.3 Hardware support for applicative languages

A simple syntax of a functional language can be given as:

```
E ::= identifier |  
      lambda identifier . E |  
      @ E E
```

which allows us to

- introduce names for objects
- use λ -abstraction
- apply one function to another

This means that, the machine should be able to execute a rich variety of application programs efficiently [51].

1.3.1.4 Variable length objects

The garbage collector must be able to handle variable length objects. If a storage manager allocates/reallocates variable sized cells, it is possible that a cell can not be allocated because no free block is large enough. This is resolved by compacting all the cells, so as to produce a large contiguous free space from which to allocate new cells [43].

1.3.1.5 Type checking

Type checking in hardware helps to build faster implementations of strongly typed programming languages [43].

It has been suggested by Harland [31] that a machine architecture which is designed to support **object-oriented programming** will go a long way towards satisfying these requirements.

1.4 The Object-oriented paradigm and machine architecture.

Object-oriented computing is often referred to as a new programming paradigm. The term 'paradigm' originally meant an illustrative example [9]. Sometimes (within computer science) this term is used as the basis for classifying programming and problem-solving methods [70]. Other programming paradigms may include the imperative-programming paradigm (Pascal or C), the logic-programming paradigm (Prolog) and the functional-programming paradigm (ML). In this sense object-oriented programming is new paradigm [9].

An object-oriented system encourages a view of the world in terms of systems of objects. In the object-oriented programming style a system is described as a collection of objects. An object is best defined as a collection of private data and public operations, or a package of information and descriptions of its manipulation. The data in an object are stored in variables. Every object has the possibility and the responsibility to maintain its own local data in a consistent state [1].

1.4.0.1 Data abstraction

Generally the definition of an abstraction is defined by Blair, Gallagher, Hutchison and Shepherd as: ‘the process of formulating generalised concepts by extracting common qualities from specific examples’ [8].

Data abstraction is generally recognised as a major step towards more structured programming. In terms of computing, data abstraction provides the starting point of object-oriented computing. The concept of data abstraction is closely related with:

- The system should be decomposed into conceptual entities. This means the breaking down of complex systems into a number of self-contained entities or modules. All information relating to a particular entity in the system is held within that module, and that module will contain all the data structures and algorithms required to implement that part of the system.
- Internal details should be hidden.

The abstraction provided by this process of modularization and information hiding is at the heart of the object-oriented approach [8].

Support of data abstraction is a necessary but not sufficient condition for a language to be object-oriented. Object-oriented languages must additionally support both the management of collection data abstraction, and the composition of abstract data types through an inheritance mechanism [69].

1.4.0.2 Communications

In object-oriented system messages are the basic mechanism for communication. The computing is performed at the level of passing messages among collections of objects.

The only way in which objects can interact is by sending messages to each other, (A message can be viewed as a procedure call or a specification of one of an object's manipulations, the difference being that the change in control flow is under the control of the receiving object). A message consists of an operation name and a collection of arguments. Sending a message to an object is referred to as performing an operation on the object. Such a message is in fact a request from the sender for the receiver to execute a procedure.

The syntax for message sending differs from system to system. In general the message passing form have the following components:

```
[ receiver      method_request:some_parameters]
```

The requester object sends a message to the receiver object asking for the specified method to be invoked. The requester is not aware whether the object can respond to the request or how the request will be carried out. It simply asks the object to attempt to fulfil the request. If the receiving object does not know how to respond to the request, it sends a 'method unknown' to the requestor (an error).

1.4.0.3 Methods

Procedures such as those mentioned in the previous paragraph, which are executed in response to a message, are called methods. The receiver decides whether and when it executes such a method. In general, the sender of the message can include some parameters to be passed to the method and the method can return a result, which is passed back to the sender. In this way objects can cooperate and communicate. Interaction between objects can only occur according to this precisely determined message interface.

1.4.0.4 Encapsulation

A very important principle is that one object's variables are not directly accessible to other objects, they are strictly private. This principle is called encapsulation.

1.4.0.5 Classes and Instances

Classes are themselves objects in the system, and provides one or more methods as an interface. One method which is always provided by a class is the method 'new', which when invoked creates *instances* of a specific class. Objects created from a particular class are referred to as instances of that class, so all instances of a particular class, even though they are not identical, exhibit common behaviour [8]. All objects of a given class use the same method in response to similar messages.

1.4.0.6 Inheritance

Inheritance is fundamental to the object-oriented Paradigm. The inheritance concept is derived from Simula. When a class Y inherits from class X, class Y now has, by inheritance, all the features of X. Thus Y is an X. Y can also be more than an X, but in addition to whatever else it may be it is also an X. This inheritance mechanism constitutes a very successful way of incorporating facilities for code sharing. The implementation can profit from code sharing by producing more compact code, occupying less computer memory [1]. At each level of inheritance an algorithm searches for a message with a specific selector in a class method dictionary. That is, the meaning of any specific operator 'message' must be efficiently and dynamically determinable for any particular type of object, so that search time should effectively be reduced. In Smalltalk [26], inheritance is primarily a mechanism for building more complex objects out of simple ones.

Objects are grouped into classes, which form blueprints for the creation of new

objects. In the Object-oriented paradigm, class is a property of objects [69]. A class provides a set of methods that describe what happens when its instances receive messages.

1.4.0.7 Polymorphism

Polymorphism is one of the most characteristic features of object-oriented system. It is informally defined as there may be a one to many mapping of a method name on implementations. For example, a method defined on one particular class is automatically defined on all its sub-classes.

Polymorphism, is a powerful technique with more general application. Modern functional languages such as ML [29], Miranda [66], Haskell [67] incorporate polymorphism [8].

Different objected-oriented programming languages use a variety of mechanisms to describe object creation.

The most important contribution of object-oriented programming in the direction of better software development methods stems from the fact that it is a refinement of programming with abstract data types [58]. It encourages the grouping together of all the information pertinent to a certain kind of entity and it enforces the encapsulation of this information according to an explicit interface with the outside world. The two important quantities of the object-oriented paradigm are **adaptability** and through its inheritance mechanism **reuseability** [1].

1.4.1 The Lingo Language

Lingo [32] is an interactive object-oriented language similar in concept to Smalltalk. The **Rekursiv(sic)** is a hardware platform designed specifically to support object-

orientation. A Lingo program is an environment composed of objects that interact only by sending and receiving messages. The implementation of Lingo on the Rekursiv is efficient; the Rekursiv's designer Harland [31] has stated that an objective of his work was to provide a processor that would efficiently execute Lingo, and other highly expressive languages by reducing the 'semantic gap'. The programmer implements a system by describing messages to be sent and describing what happens when messages are received. The fundamental ideas of objects, messages and classes came from Simula [7] and the message passing terminology from Smalltalk [26].

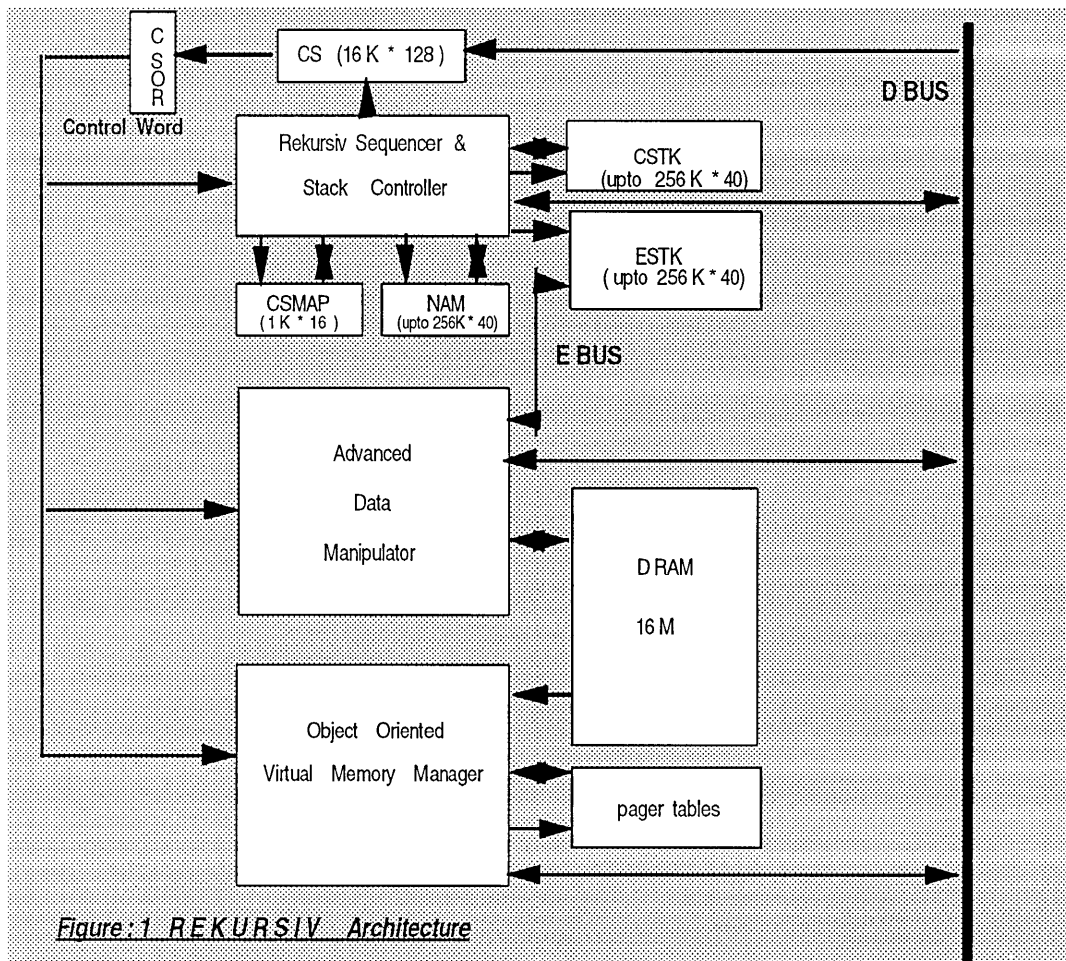
1.4.2 The Rekursiv

For a completely integrated objected-oriented storage system abstract operations should be single machine instructions, not sequences of low-level opcodes directly manipulating the store [31]. In a von Neumann machine, low level opcodes do not manipulate abstract objects in the store; the Rekursiv [32], on the other hand, provides higher level opcodes for object manipulation. Furthermore, an ideal machine to support functional programming should provide the hardware support for sophisticated storage management [31, 68]. Providing an object-based persistent storage system (see below), the Rekursiv appears promising in this regard [31].

The particular form of the Rekursiv available for this study is known as Hades board. This unfortunately did not have its own disk processor (DP) to enable the persistent object store. Instead, software (called KONTROLLER) on the host Sun provided the functionality of a disk processor but at a very reduced bandwidth in comparison to what would be achieved with a direct connection to the Rekursiv. KONTROLLER maintains the persistent store by communicating on the VME bus with the Rekursiv and if necessary retrieving object images from disk by using Unix file system calls.

The Rekursiv architecture represents objects directly, maps them into a persis-

tent store, automatically swaps them in and out of memory, and performs range checking (see below in 'Objects on the Rekursiv'). It is a tightly coupled cluster of processors (see **Figure 1**), each processor is optimised for a specific kind of operation (ie one for type checks, one for index checks etc). The internal operations of the various processors take place in parallel.



Microcodeability allows high level operations to be presented as single instructions. To be able to provide for a very high level instruction which undertakes messaging, direct support must be provided for method lookup. Searching in method lookup is complicated by the fact that it is possible to inherit methods from an objects's superclass, and from the superclass of the superclass, and so on upwards through an inheritance chain.

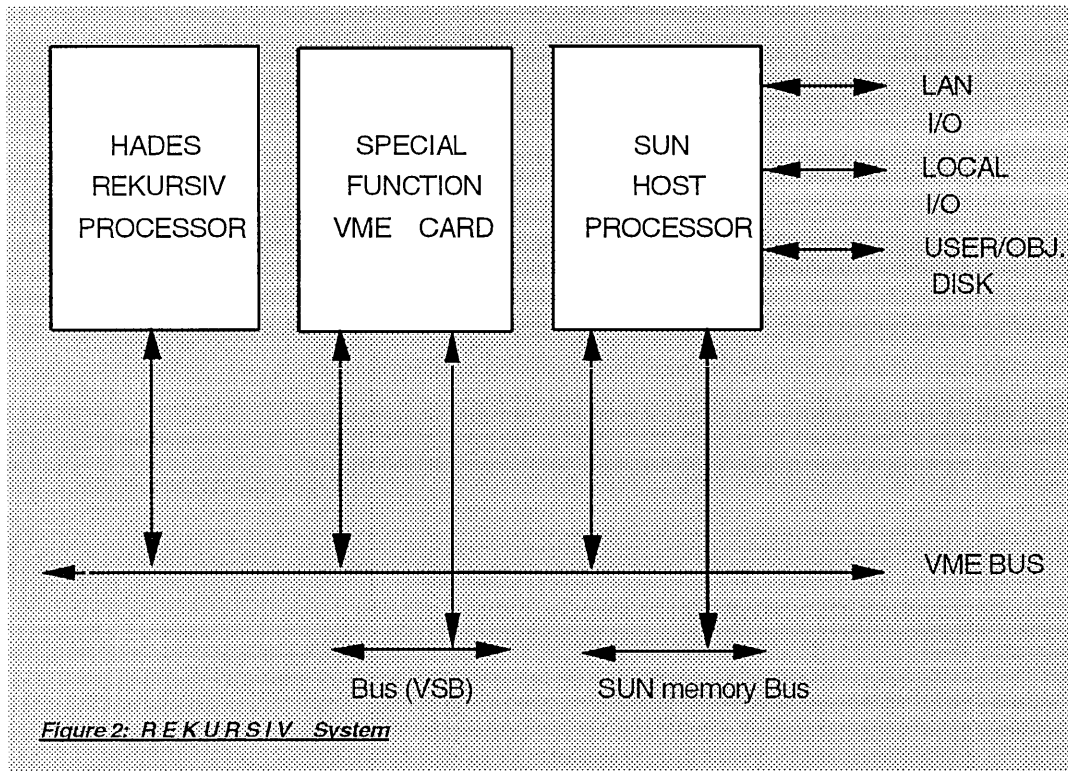
The **Rekursiv** is therefore a microcodeable processor that utilises a persistent object memory. Objects in memory are addressed by unique object identifiers and not by any absolute or relative physical address. The virtual memory scheme is not based on the conventional fixed size partitions of a paged memory. Instead the memory manager can employ strategies to ensure that the most commonly accessed objects remain resident in main memory.

The **Rekursiv** differs from other microprogrammable processors in allowing the microsequencer access to a stack. This allows high expressivity at the microcode level since recursion and procedure calling are possible.

Because everything is an object, an object is identified by its object number. The details of object type and size are hidden behind (in Lingo) a small number of primitively implemented methods such as **ObjectClass** and **ObjectSize** on class **Object**.

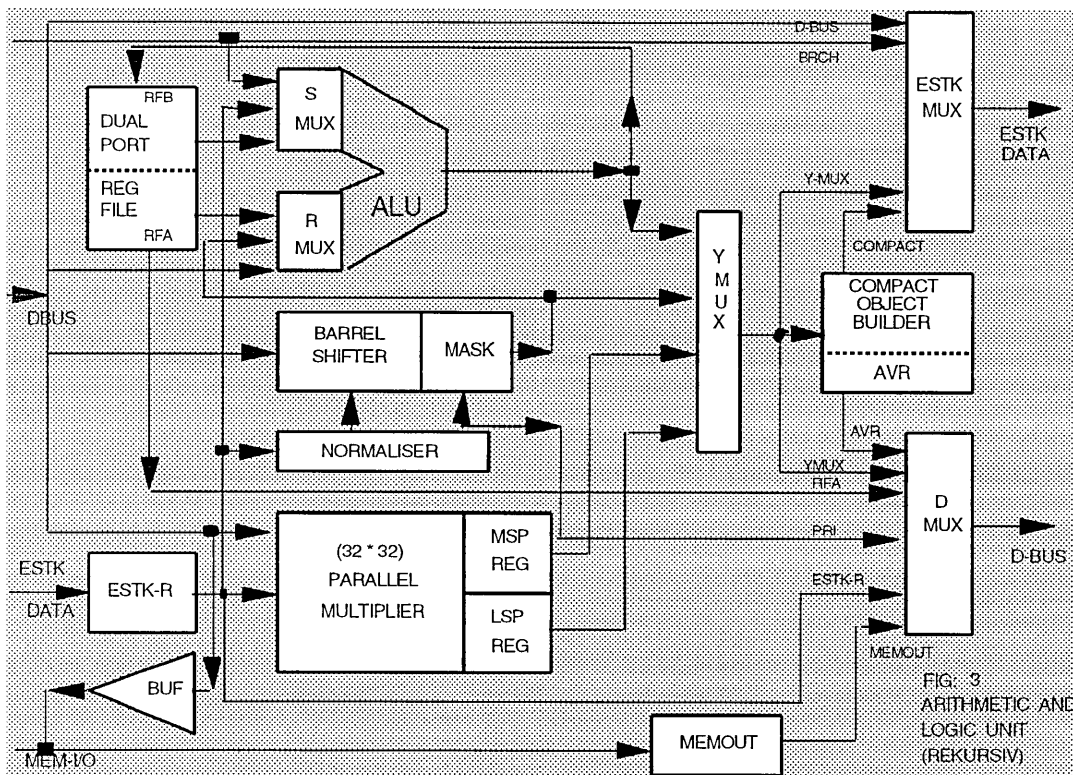
Rekursiv is a 32-bit machine. It uses a 40-bit wide bus to accomodate the object identifiers. Object identifiers are 40-bits wide, and have the top bit (bit-39) set to allow easy distinction between object identifiers and other items.

Rekursiv objects are allocated memory contigously [31]. Its CPU has been integrated with 16M bytes of main memory(object store). The Object store is in two parts: RAM and on Disk [31]. The following figure (**Figure:2**) illustrates the input and output routed through the Sun(host machine) and the communications to the **Rekursiv** and other special function cards.



This means that for the HADES Rekursiv all I/O is effected by routing request via the VME bus to a process running on the Sun which ultimately performs the I/O.

The ALU as shown in **Figure-3** performs only 32-bit arithmetic and logical operations and it can access and drive only the lower 32 bits of the bus. Object numbers, which carry information encoded in the topmost 8 bits, can be created only by the memory management hardware. (There are tables to hold an object size, an object type & class, an objects current address in memory, and the first word of the object).



In the Rekursiv there is hardware for generating certain condition codes to assist with the process of garbage collection [31]. Rekursiv presents an opportunity to provide ‘garbage collection’ in hardware. It is microcoded as part of the object-oriented instruction set. So the basic strategy for finding out which objects may be kept or which are no longer required is handled directly by Rekursiv’s hardware.

The object store deals with objects directly. It can delimit objects in memory, determine whether they are in-core without causing them to be brought in and determine their relative age and status and do so in a very simple and transparent manner (see below in ‘Objects on the Rekursiv’), to make memory management an effective and efficient process [31].

As opposed to a conventional machine, where memory is considered to be a linear array of equally sized words each with a unique address, objects in store on the Rekursiv can be of any size and are addressed by a unique (hardware assigned) object number.

1.4.2.1 Memory management

The main memory addressing system of Rekursiv has two primary functions.

- To compute the address of an object body entry and the objects current main memory start point.
- To manage the allocation of space in main memory for newly created or swapped-in objects.

The programmer requests a particular object by issuing that objects unique identifier.

1.4.2.2 Objects on the Rekursiv

Central to the operation of the REKURSIV is its handling of objects. These can be up to $2^{24} - 1$ words (40 bit) large and are addressed by 40 bit quantities known as *object identifiers*. Object identifiers are (hardware) distinguishable from other quantities by a coding in the two highest order bits of a forty bit word.

The object identifier is the only means of accessing an object, and so it can be thought of as the *address* of the object in some large virtual store. The next sub-section describes briefly the mapping of object identifiers (viewed as virtual addresses) to physical storage addresses.

Objects are stored in memory along with a fixed sized header, each of whose component words are deposited in accessible registers whenever the object is addressed. The information in this header includes a forty bit word denoting the type of the object. Since this word can itself be an object identifier, object-oriented language implementation is facilitated by using this word to denote the class of the object. Method determination, which in general involves ascending an inheritance

tree can then be microcoded since the ‘type’ information for an object is hardware accessible. (Method determination and despatch are important features since they are the major actions involved in the execution of a pure object-oriented language program.)

Lisp machines [71] deal with this problem slightly differently by tagging objects with their type. The tags are not, however, accessible simultaneously with the body of an object.

The other header information allows the hardware to check that accesses to the component words of an object are safe.

1.4.2.3 The virtual memory

Clearly at some point objects must be stored in a real memory constructed from a linear array of RAM cells. To effect, in hardware, the translation from an object identifier to a location in real memory the REKURSIV makes use of ‘pager tables’. These take the place of the page table of a conventional paged virtual memory and essentially associate virtual addresses (object identifiers) with object information (real address, modification bits and so on).

A difference is that the pager tables are indexed by the lower order sixteen bits of an object’s identifier. This restricts the number of ‘active objects’ that the REKURSIV can maintain to 65536. Any more and it must clear an entry in the pager tables. This partially corresponds to the page fault of conventional systems but this fault may occur even though there is plenty of space left in physical storage.

On a pager table full fault, the decision as to which entry to replace (to make room for the object whose addressing caused the fault) is straightforward and is merely determined by the low order sixteen bits of the newly referenced object.

The decision as regards the writing to disk (*‘squeezing’*) of the image of the

displaced object (which is still in RAM, but not recorded in the pager tables) is based on data held in an object's header describing if it has been modified, or if it is new.

The strategy embodied in the supplied controlling software for the REKURSIV (KONTROLLER) squeezes new objects that are displaced from the pager tables (and then triggers a garbage collection of physical memory). This strategy can be disconcerting when many new, but small, objects are being produced as **squeeze** faults (and their attendant garbage collections) occur even though physical memory is underused.

1.5 Some of Harland's claims

Harland attempted to design a more powerful and more secure architecture than conventional von Neumann architecture (vNA) machines, which provides a combination of hardware and software which provide:

1. Performance, supporting a large number of abstract data types.
2. Expressibility, to support an arbitrary level of abstraction.
3. Security, to guarantee the semantic integrity of data

Harland attempted to meet this challenge by defining a computational model that is powerful enough to express solutions to a wide range of problems:

1. design a new language in which every thing is a value, dynamic and arbitrarily manipulable. In principle all aspects of a system's working should be manipulable via the same computational mechanism.
2. design a machine structure which

- Efficiently implements dynamically typed languages:

The majority of programming languages employ types merely as a means of specifying to their compilers how storage is to be arranged and to provide just sufficient information to enable a compiler to carry out rudimentary type checking. In these load and go systems type checking is a static process and it is all over before program begins its run. Type information is not generally preserved for use during execution and types themselves are not manipulable objects.

On a conventional vNA machine, dynamic type checking imposes a considerable overhead, because the code which performs each check must be inserted into the object code, slowing down the program even when there are no errors. Although many compilers can insert such checks, they tend to do so only as an option, so what we need is high-level machine on which such type checks and range checks must be carried out automatically.

- Powerful abstraction mechanism:

If an abstraction mechanism itself is to be dynamically modifiable then its primitive operations should also be implemented in a higher order manner, so that they can recognise new or changed semantics. That is, the notion of 'evaluate' and 'application' should be dynamically driven too.

- The elements of a heap traditionally refer to one another in terms of addresses and during garbage collection. Most language implementations for von Neumann machines incorporate their own individualistic garbage collection routines for private heap storage. So they do not provide very good support for heap structure simply because they implement too low a level of storage abstraction, and this is one reason why it is so difficult to integrate languages on such machines.

- Should have one level storage system so that ‘persistence’ is a natural attribute of all data.
- Permit high-level functions to be called as efficiently as low-level jumps.

Machines with low level semantics (RISC) can not manipulate abstract structures directly and so can not provide ‘secure’ systems in which data integrity is guaranteed. The solution proposed by Harland is to use an object-oriented machine. This machine, called Rekursiv, makes use of VLSI technology in a different way to RISC architectures.

Harland argues that to manipulate abstract structures efficiently a high level instruction set is required which encapsulates the detailed structure of the data within an object. To provide a high level instruction set, he proposes a microcodeable architecture. The microcodeable code itself has a high degree of expressivity available through recursion and the use of type checking hardware.

Rekursiv provides an Object-oriented store, full persistence of all types of data, and the possibility of microcoding an instruction set which eliminates the semantic gap entirely by directly implementing the primitive operations of the object-oriented programming system environment.

With Object-oriented programming, large systems can be divided naturally into coherent modules which can be developed and maintained separately.

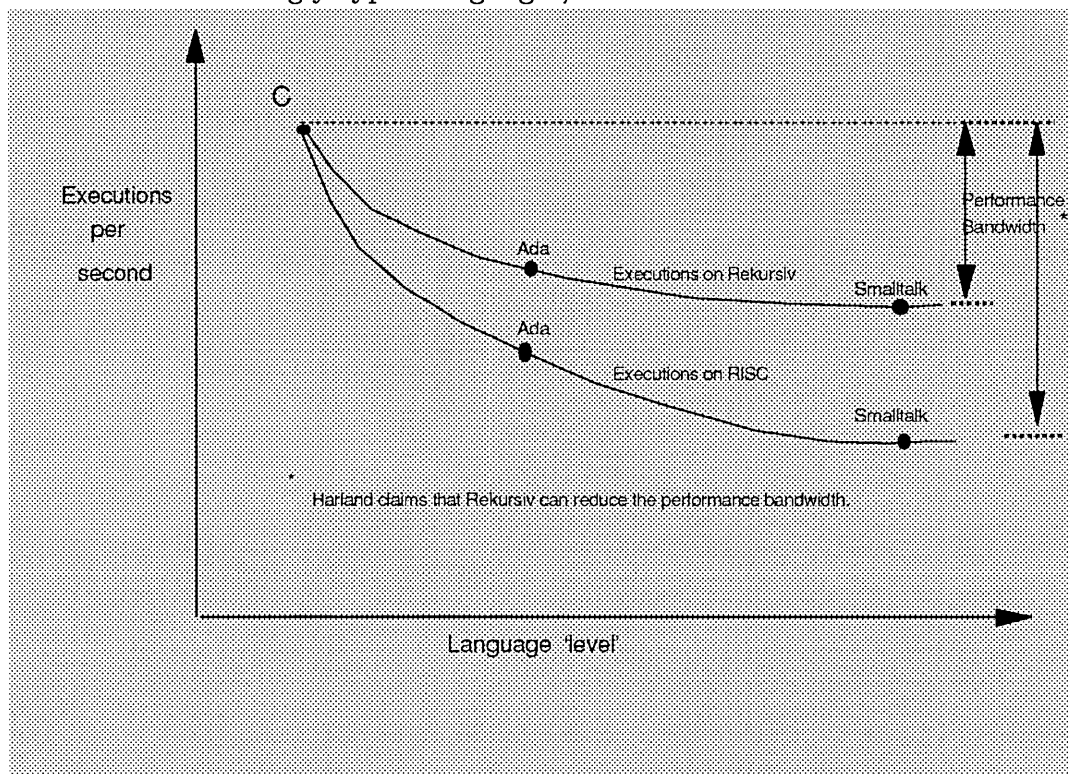
Harland makes the following claims for his Rekursiv:

- It is an ideal engine for Knowledge representation and symbolic computation.
- Garbage collection is done in hardware.
- It directly implements the Object-oriented paradigm.
- Objects can be passed from one process to another or from one language to another.

- Type checking is performed in hardware.

The Rekursiv processor's memory is heavily differentiated into many distinct banks of memory. There are separate memories for stack, code and data. Each memory is directly accessible and manipulable by the associated processing elements, independently of the main system bus, which reduces the need for bus traffic. Because it is a tightly coupled cluster of processors, the overhead of the communication protocol is overcome by using a single control word to synchronise all the elements.

In terms of strongly typed languages, Harland's claims can be shown as:



Strongly typed languages impose security by limiting the range of things that can be expressed, but limiting expressivity restricts the range of potential applications. LISP for example is expressive and has an enormous potential range of applications, but it runs slowly on conventional computers.

The above figure shows that if the performance of the programming language 'C' is more or less same on both architectures there will be a considerable gain in

performance when executing Smalltalk type languages on the Rekursiv. This means that, if the execution of a 'C' program takes one second on RISC then it may take similar (one second) execution time on Rekursiv. But if a Smalltalk program runs 3 times per second on RISC then it will run more than 3 times per second on Rekursiv. That is, the Rekursiv reduces the performance bandwidth.

1.5.1 Features to be investigated

The following features of the Rekursiv have a bearing on this project:

- Typechecking in hardware
- Persistence
- Garbage collection

1.6 Aim.

The aim of the project is to investigate the capabilities of an object-oriented architecture in providing a general platform for the implementation of functional languages.

In order to meet this aim, the following objectives have been identified:

1. To generate an implementation of a functional language using the Lingo language on the Rekursiv.
2. Performance analysis on various implementation techniques.
3. To identify the major issues to be investigated in terms of general implementation strategies in an object-oriented environment. As a result of this, it will be possible to determine those parts of the implementation that are critical

for efficiency. It will then be possible to focus attention on those critical areas with a view to improving their efficiency.

4. The determination of the contribution to performance resulting from the underlying Rekursiv architecture by comparing the Rekursiv implementation against implementations using other object-oriented languages on a conventional processor.
5. The contribution of the style of object-oriented implementation to performance.
6. To determine to what degree built-in features of the implementation language (eg garbage collection) can be exploited.

1.7 Summary

In this chapter an informal introduction to functional languages and functional programming concepts has been presented. The object-oriented paradigm has been considered as an alternative implementation approach for functional languages, particularly the Rekursiv [31] processor using the object-oriented language Lingo.

The convergence of ideas in the hardware field and in the software field is of growing interest for the implementation of functional programming languages. The designer of the Rekursiv, David Harland claims that this offers the potential for the efficient evaluation of functional programs on the Rekursiv as a result of the reduced performance bandwidth; resulting in a considerable gain in performance when executing Smalltalk type languages (for example 'Lingo') on the Rekursiv rather than on a conventional machine.

The next chapter introduces the λ -calculus and its use in functional language implementations.

A brief study of the many approaches to functional languages implementation like **graph reduction**, **environment-based** and **data flow** techniques is presented, and an object-oriented ‘active graph’ implementation is proposed.

Chapter 2

The implementation environment

This chapter is mainly concerned with implementation techniques of functional programming languages. Because most purely functional programming languages are a version of the λ -calculus, so their programming can be done in a purely descriptive fashion, some ideas of λ -evaluation, substitution, reduction and conversion are explained briefly in the first part. This is followed by a discussion of the mechanism of graph reduction which is used for the implementation of functional languages. Other implementation techniques are reviewed.

2.1 The λ -calculus: Introduction and its syntax

The λ -calculus provides a method of representing functions and the conversion rules for syntactically transforming them. For example a successor function on integers may be expressed as:

$$\lambda x. 1 + x$$

The λ notation as devised by Alonzo Church [14] is read as ‘the function of’ and dot (.) as ‘which returns’. The variable ‘x’ after the ‘ λ ’ is called a bound variable (discussed below) and the expression to the right of the dot is called the body of the abstraction. The body of the abstraction may be extended to the right and may be any valid λ -expression. For example, In ‘ $\lambda a. \lambda b. a$ ’ the body ‘ $\lambda b. a$ ’ is a valid λ -expression. Extra brackets may be used to clarify it as ‘ $(\lambda a. (\lambda b. a))$ ’. Note that the λ -expression of this form can not be recursive as there is no associated function name to refer to. We can write equations defining functions in which just one identifier appearing on the left hand side of the equation serves the job of naming that function, for example

$$\text{def Succ} = \lambda x. 1 + x$$

In the case of recursion, the usual ‘loop counting’ is replaced by recursive calls. Recursion can be simplified by abstracting at the place where recursion takes place in a function and then passing the function to itself [55].

In order to call a function repeatedly, we must name the function by using ‘def’ as:

$$\text{def } <name> = <expression>$$

Recursion by passing a function to itself can be given as:

$$\begin{aligned} \text{def } <name> &= \dots (<name>) \dots \\ \text{or } \text{def } f &= \dots (f f) \dots \end{aligned}$$

The right hand side represents the expression in which the function definition is repeated. For example:

$$\text{def } \underline{fact} = \text{if } x = 0 \text{ then } 1 \text{ else } x * \underline{fact} (x - 1)$$

This indicates that the recursive function ‘fact’ can be substituted at its own recursion points. Further details of how to deal with this problem are given in Chapter-4.

Apart from the capability to write a ‘program-forming program’, the most notable feature of programming in a λ -based notation is the absence of the assignment statement [24].

2.1.1 The evaluation of λ -expressions

Before describing the conversion rules of λ -calculus, the terms ‘bound’ and ‘free’ variable should be identified.

An occurrence of a variable is **bound** if there is an enclosing λ -abstraction which binds it, and is **free** otherwise. For example

$$((\lambda x. z x)(\lambda y. y x)) \text{ ————— } (1)$$

Where z and x are different. The first occurrence of ‘ x ’ is bound, and the second occurrence of ‘ x ’ is free.

A variable may be bound and free in different places in the same expression. For example

$$(\lambda x. \lambda y. (x y))$$

This has bound variable ‘ x ’, and the body expression is another function:

$$\lambda y. (x y)$$

which has bound variable 'y' and free variable 'x'. The variable 'x' is bound within the outermost λ -abstraction and free within the inner λ -abstraction.

Example (1) also illustrates how functions of more than one argument are defined in λ -calculus. Functional application is represented by juxtaposing one term with another (juxtaposition can be taken as an indication of functional application).

A clause of λ -calculus which involves the application of one term to a another is an instance of functional application. Apart from variables and constants in λ -calculus there are only two ways of forming new terms from the old ones, that is, definition of a function and its application to an argument.

A computation can be performed on a λ -expression by reducing **redexes** in it, until it is in **normal form**, (a redex is a reducible functional application expression). This can be performed by employing three rules: α -conversion, β -reduction and η -reduction.

2.1.1.1 Conversion rules

These rules that define the ways of interpreting a λ -expression.

- $(\lambda x. E) \xrightarrow{\alpha\text{-conversion}} (\lambda y. [y/x] E)$
- $(\lambda x. E) v \xrightarrow{\beta\text{-reduction}} [v/x] E$
- $(\lambda x. E x) \xrightarrow{\eta\text{-reduction}} E$

Where the notation of the type $[v/x]X$ is used to mean 'the expression X with every unbound occurrence of the variable x replaced by the value v'.

2.1.2 Different evaluation paths

The evaluation of a λ -expression proceeds (primarily) through β -reduction (that is the replacement of a bound variable with an argument in a function body). In functional languages, β -reduction is the closest analog of assignments [74]. There are three distinct evaluation mechanisms which define different orders of evaluation of sub-expressions (i.e in what order to apply β -reduction) and hence different evaluation paths. These are

- Normal order
- Applicative order
- Lazy evaluation

The Church-Rosser theorem [14] shows that every expression has a unique normal form. An expression containing no redexes is said to be in normal form.

Since our main aim is to get the normal form from the input expression the following points will help to choose an appropriate evaluation mechanism.

- applicative order may not terminate (see example below which covers all the following listed points)
- normal order may repetitively evaluate argument expressions, because the left most reduction is always done first if there is a choice.
- normal order may not terminate
- there is no way of deciding whether or not the evaluation of an arbitrary expression will terminate.
- if a normal form exists then it is unique and it will be reached by normal order evaluation (second Church-Rosser theorem). If any evaluation order will terminate then normal order evaluation is guaranteed to terminate.

Example :

The following example illustrates normal and applicative order reductions clearly. This also shows how an expression terminates using normal order, but will not terminate with applicative order.

Consider the following expression:

$$\overbrace{(\lambda x. y) ((\lambda x. x x)(\lambda x. x x))}^{\text{original expression}}$$

$\underbrace{(\lambda x. y)}_{\text{function application}} \quad \underbrace{((\lambda x. x x)(\lambda x. x x))}_{\text{argument expression}}$

The first part of the original expression is the functional application and the second part is the argument expression.

To evaluate it, using applicative order reduction, the argument expression must be simplified first. The argument expression which is to be evaluated first is:

$$((\lambda x. x x)(\lambda x. x x))$$

The evaluation of this expression will not terminate. Since $((\lambda x.xx)(\lambda x.xx))$ reduces to $((\lambda x.xx)(\lambda x.xx))$ and does not terminate in a *normal* form. ie

$$(\lambda x. x x)(\lambda x. x x) \xrightarrow{\beta} (\lambda x. x x)(\lambda x. x x)$$

Applicative order thus does not terminate.

With normal order reduction the situation is different. In normal order we first substitute the bound variable (which is 'x' in this case) in the function with the unevaluated argument. That is reducing the application without evaluating the argument. Normal order reduction immediately terminates with the normal form 'y'.

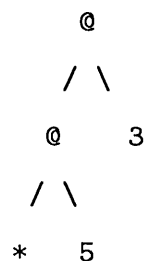
Thus if an expression is reduced using both evaluation orders and both terminate, then they produce the same final result, even though normal order may lead to multiple evaluation of the argument.

2.1.2.1 Lazy evaluation

The most important step forward in evaluation strategy is ‘laziness’, first introduced by Wadsworth [75]. This achieves normal order evaluation without causing a large increase in the amount of work to be done. The extra reductions introduced by normal order reduction are really just repeated calculation, requiring any work in duplicate text to be performed more than once. This problem can be solved by representing the λ -expression as a graph which represents the duplicate text by a shared subtree. Once the subtree has been reduced, that reduction will never be performed again.

Lazy evaluation evaluates an expression only when its value is needed. For a purely functional program, a lazy reducer performs no more (and possibly fewer) reductions than an applicative order one. Thus an expression is only evaluated when it appears in the function position of a functional application. In addition, after evaluation all copies of the expression are updated with its new value.

Consider the body of $(\lambda x. * x 3) 5$, that is, $\xrightarrow{\beta} * 5 3$



We can define $\lambda x. E$ as a special type of node, a λ node, with bound variable associated with its left subtree and the body with the right subtree. Here '@' denotes the application of left subtree to right subtree. Consider an example of β -reduction on

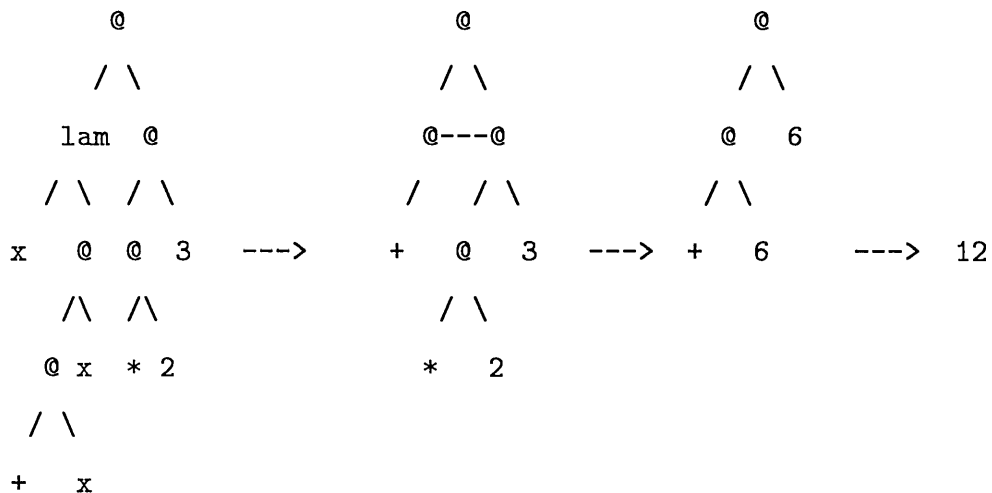
$$(\lambda x. + x x)(* 2 3) \xrightarrow{\beta} (+ (* 2 3)(* 2 3))$$

here we evaluate the first $(* 2 3)$

and, as we shall see, this value replaces both of the occurrences of the argument expression which gives

$$\xrightarrow{\beta} (+ 6 6) = 12$$

This can be represented as.



Where 'lam' denotes a λ -node, and '- - -' denotes sharing.

The lazy evaluation combines the best features of normal order (a guaranteed normal form if it exists and delayed evaluation of arguments) with those of applicative order (efficiency due to only evaluating multiple copies of an argument once).

Functional languages like **ML**[3] and **Hope** [11] are *strict* [43]. A function is strict if it is sure to need the value of its argument. In terms of reduction order, strict semantics means applicative order reduction, that is, reducing the argument before reducing the application of the λ -expression [43]. On the other hand, **SASL**, **KRC**, **LML**, **Miranda**, **Orwell**, **Ponder**, **Haskell** are all lazy [43, 50].

We may conclude from this discussion that, in order to avoid redundant evaluation and to guarantee ‘safe’ termination properties, all implementations of functional languages should be lazy [22]. However, there may be some overheads (for example, the exact time at which the arguments will first be needed and hence evaluated) associated with the passing of parameters lazily. For this reason, some functional languages are defined to have a ‘stricter’ definition of semantics.

Even though languages like **ML** and **Hope** (as mentioned above) have a stricter definition of semantics, they support lazy evaluation where it is explicitly requested by the interpreter, particularly in data constructors [43]. For example, in **Hope** a single lazy constructor function ‘lcons’ is used to support lazy evaluation and infinite data structure (lists).

2.2 Implementation techniques

The implementation of functional languages is generally described in terms of an abstract machine [21, 40] reducing either the source functional program or a compiled version of a program. The technique of reducing the source program is based on an implementation where an expression is not shared among multiple references, so subexpression may be evaluated many times. An example of this technique is Backus’s metalanguage FFP, which allows massive parallelism to be exploited [4].

In the technique of reducing the compiled version of the program, a slightly more complex representation of an expression is used. In this technique expressions

are represented as graphs, which allows references to a common sub-expression and common sub-expressions may be shared. Thus graph reduction is a process in which expressions are represented as graphs and graphs may be shared. Sharing is important to the performance of a sequential evaluator, because it avoids re-evaluation of copies of common expression [61].

2.2.1 Graph reduction

Graph reduction gives a ‘good fit’ with lazy evaluation [43].

In imperative languages, the order in which statements are executed usually affects the result. In functional languages the choice of order makes no difference to the result (provided it terminates).

The computation in graph reduction can be viewed (see below) as a controlled deduction rather than a sequence of apparently meaningless state changes [24]. Perhaps this is the most fundamental aspect of *novel architecture* work.

In graph reduction, the data structure is modified (reduced) until it becomes the desired result. That is, for a β -reduction step, each occurrence of a bound variable is eliminated by substitution of the corresponding argument expression found in an application. The data structure (in the form of a graph) consists of a number of nodes, each of which can contain a number of other nodes as well as atomic values (or leaves). The evaluation strategy is to look at the leftmost function name in the expression: If it is a user defined function then it is replaced by its definition (body) and the result will still be an expression, but if it is a primitive function like plus or times, then the indicated operation can actually be performed, replacing the function application by its result. This reduction process is repeated until there are no more redexes, when the expression is in normal form.

For example, $((2*3)+(4*5))$ contains two redexes; $(2*3)$ and $(4*5)$. These may

be reduced to 6 and 20 respectively, and the original expression reduced to $(6+20)$, which is further reduced to the normal form 26. In this way the graph becomes reduced until no further reductions can be performed.

Graph reduction is also known as **copy-based** implementation; the application of a function to an argument causes the graph of the function body to be copied with the appropriate argument substituted [22]. This means that the evaluation of an expression is only performed when its value is demanded (lazy evaluation), and function application takes place as soon as the argument of that function becomes available [22].

A further advantage of graph reduction is that normal order evaluation is easy to express and relatively efficient to implement.

Most of the current and proposed functional machines use some form of graph or list structure to represent the program. Of these the most notable are the:

- G-Machine, devised by Thomas Johnsson 1984 & Lennart Augustsson 1984, Peyton Jones 1987 has incorporated this idea in a graph reducer that uses supercombinators (see below) to increase execution speed [61, 43].
- SKI- Reducer, devised by Turner. For combinatory-based implementations (see below) the seminal reference is Turner [65].
- TIGRE: Threaded Interpretive Graph Reduction Engine, devised by Koopman 1989, for executing combinators (see below) in a pure graph reduction style [49].
- TIM: The three instruction machine devised by Wray and Fairbairn 1988. The TIM is remarkable as it achieves respectable efficiency without the many optimizations required by the G-machine [2].
- G-TIM: Improving the TIM, devised by Guy Argo 1989, An improved TIM

which is more amenable to optimization based on sophisticated compile time analysis [2].

All of the above mentioned implementations are based on graph reduction using standard or supercombinators (see below) and can be viewed as interpretive, because the reduction and transformation of their graphs can be seen as interpretation of the graph. For example in the G-machine [40] graphs are transformed into abstract machine instructions, which can be rendered into machine code. Current implementation methods used in practice for functional languages usually correspond more or less closely to graph reduction.

Graph reduction is a particularly natural vehicle for supporting lazy evaluation in functional languages [22].

2.2.1.1 Use of Combinators in graph reduction

Most recent graph reduction implementations are based upon:

- Standard combinators [13, 65],
- Supercombinators [37], or
- Categorical combinators [51].

All these approaches have the same common property ie the selection of the next reduction step which is dynamically derived from the current expression form at each stage in a reduction sequence.

Standard and supercombinator implementations are based on the fact that all of the variables in a λ -expression can be abstracted by transforming it into a sequence of combinators (ie constants or expressions containing no variables at all). By removing

all bound variables the resulting expression can be represented as a graph, and so evaluating the expression becomes a process of **graph-reduction**.

Standard and supercombinator approaches adopt a two-dimensional graphical representation of expressions. Their implementations are described in chapter-3. Categorical combinator reduction is based on a sound mathematical foundation, which facilitates a good degree of optimization. Its operational semantics are very much like that of the SECD (environment based) machine [22]. Unlike standard and supercombinators categorical combinators are not fully lazy. The categorical combinator approach is related to an environment based implementation (see below), and so is not readily comparable with the other combinators. In addition, they have not matured into a mainstream implementation method, and so are not considered further.

In most graph reductions the traversing of the graph's left spine (stack unwinding) and the case analysis of nodes are costly in terms of performance. If the costs are reduced, significant speedups may be possible [49].

2.2.1.2 Active graph implementation

In this thesis the idea of a combinator graph data structure is extended to that of an **active graph**; ie each combinator is represented by an object in the object-oriented paradigm and contains the code for its own reduction. The active graph combines the data structure and the algorithm required to reduce it.

Combinator graphs or the λ -lifting technique introduced by [43, 40] are particularly attractive for such an object-oriented implementation (see chapter-4).

2.2.2 Other Techniques

We have discussed one way to deal with the bound variable problem. That is the way of performing β -reduction, which performs the substitution which replaces the parameter reference by its argument (i.e. copy-based). In other approaches the most notable is the environment-based one. In this approach, we leave the formal parameter reference (formal parameter denotes the variable just after the λ -notation) as it is but make a record of what value it denotes in a separate data structure [22].

One of the motivating factors in these approaches is the ‘problem of bound variables’ ie identifying the existence of bound variables in the function bodies which leads to the appropriate technique. For example, consider the following λ -expression

$$\begin{aligned} (\lambda x. (\lambda y. + x y) 1) 2 &\xrightarrow{\beta} (\lambda y. + 2 y) 1 \\ &\xrightarrow{\beta} + 2 1 \end{aligned}$$

the body $(\lambda y. + x y)$ is different for different bindings of x . Since the variable ‘ x ’ occurs free in $(\lambda y. + x y)$, so it is β -substituted with the outer most argument ‘2’.

In *environment-based* schemes, an environment is an association or binding of variables with values to be substituted for them in the evaluation of a expression. It is a technique which provides for sharing and delaying of the substitution of values for variables [60]. An environment based scheme is a variation of sequential code in which a graph structure is used to represent the program. In graph structured machine code the environment is kept in a separate structure such as a display or association list.

2.2.2.1 SECD machine

The first environment-based implementation used the SECD machine. It was introduced by Landin in 1964. It is based on an abstract architecture for the applicative

evaluation of λ -expressions [33].

The SECD machine is a stack-based implementation and uses four ‘stacks’, labelled S, E, C and D for the evaluation of λ -expressions.

This machine can be viewed as ‘a generalisation of the abstract machine underlying implementations of imperative programming languages like Pascal’ [63]. The eager and lazy version of the SECD machine are often termed **data-driven** and **demand-driven** respectively.

In the SECD approach the code sequence for a λ -abstraction has access to an environment which contains values for each of the free variables, thus allowing a single code sequence for each λ -abstraction [43].

The combinator approach, on the other hand, supports sharing and copying. This means that, in contrast to the SECD machine, the λ -abstractions can be compiled out and a simpler machine (graph reducer) may be used to evaluate the resultant combinator code.

It was mentioned earlier that the SECD machine uses an applicative order evaluation route and so is incapable of handling infinite data structures. The SECD machine needs modification so that it evaluates in (graphical) normal order.

One basic drawback of the environment-based scheme is that in order to understand an expression fully the environment in which it was created must be consulted [24]. In an implementation which supports functions which can accept and return functions, this means that each expression must contain at least a reference to the environment in which it was created. That is, every β -substitution is to be remembered. Kennaway [24] describes this remembering process as a considerable amount of excess baggage which may be accumulated in the form of environmental entries which will never be consulted. The schemes which minimise this excess baggage tend to destroy the basic point of the environmental approach by complicating the

remembering process.

The other drawback which relates with this excess baggage problem is that it needs an efficient look-up mechanism. The larger the environment, the higher the cost of entry and look-up.

Combinator code in general is about twice as compact as SECD code. This leads to an extra storage requirement [65]. Turner [65] shows a comparison on relative performance of SECD and combinatory code reduction for a factorial function. The combinatory code occupies 13 cells while SECD code needs 24 cells.

Because of these drawbacks the SECD approach will not be considered further.

2.2.3 Data flow

Data flow is an approach to parallel processing. The fundamental principle of data flow operation [17, 64], is that an instruction is ready for execution when all its operands are available. A disadvantage of the approach is that instructions may waste time waiting on the evaluation of unneeded arguments. For example, an operator such as if-then-else, which will use only two of its three arguments, will always be forced to wait for all three to be evaluated. The data flow approach is best suited to machines which support parallel execution and thus is not of direct relevance to this project.

2.3 Summary

We have discussed the two extremes in functional language implementation, the SECD machine and graph reduction. The major difference between the two lies in the nature of their abstract machine and in the function calling semantics they support.

Field [22] has shown that there should be a close correspondance between the abstract and concrete machine architecture, so translating expressions into code which is directly executable (active graph) should result in very efficient code.

Optimized combinators like Bprime, Cprime etc (see next chapter) can be built into the code generator to improve the quality of the resulting code. Typically this is concerned with the compiled code, so that the combinators are compiled directly into executable code (active graph, see chapter-3) rather than the use of stack pushes [22].

Graph reduction was selected as an appropriate vehicle for this project. The reasons for this are outlined below:

Graph reduction has emerged as a powerful implementation model for lazy functional languages. Combinators and full laziness are two of the key techniques available for the efficient implementation of graph reduction [42]. Graph reduction is a pratical implementation technique for functional programming languages, with the important advantage of full laziness [37]. It also supports sharing in a natural way.

As we shall see (in chapters 3 & 4) our 'Alcal' implementation maps remarkably easily and efficiently onto an Object-oriented architecture, in the form of an executable graph. Furthermore in the Rekursiv the garbage collection is performed by the underlying hardware during execution. Automatic garbage collection relieves the compiler writer of the need to keep track of references within the program and to release the storage space explicitly.

The next chapter describes combinatory logic which includes combinators and supercombinators, as well as their reduction rules, identification of bound or free variables in an expression, and the technique of transforming a λ -expression into combinatory form.

Chapter 3

Combinator Reduction

This chapter's main issue is techniques for transformational algorithms. Taking in view the differences of λ -calculus and combinator calculus, the problem deals with the translation of λ -calculus expressions into combinatory form, firstly by using the fixed combinators proposed by Turner and secondly using the 'customized' Hughes-style supercombinators. The method of supercombinator abstraction is derived from Hughes' original algorithm. Finally the advantages of this new improved supercombinator abstraction are discussed.

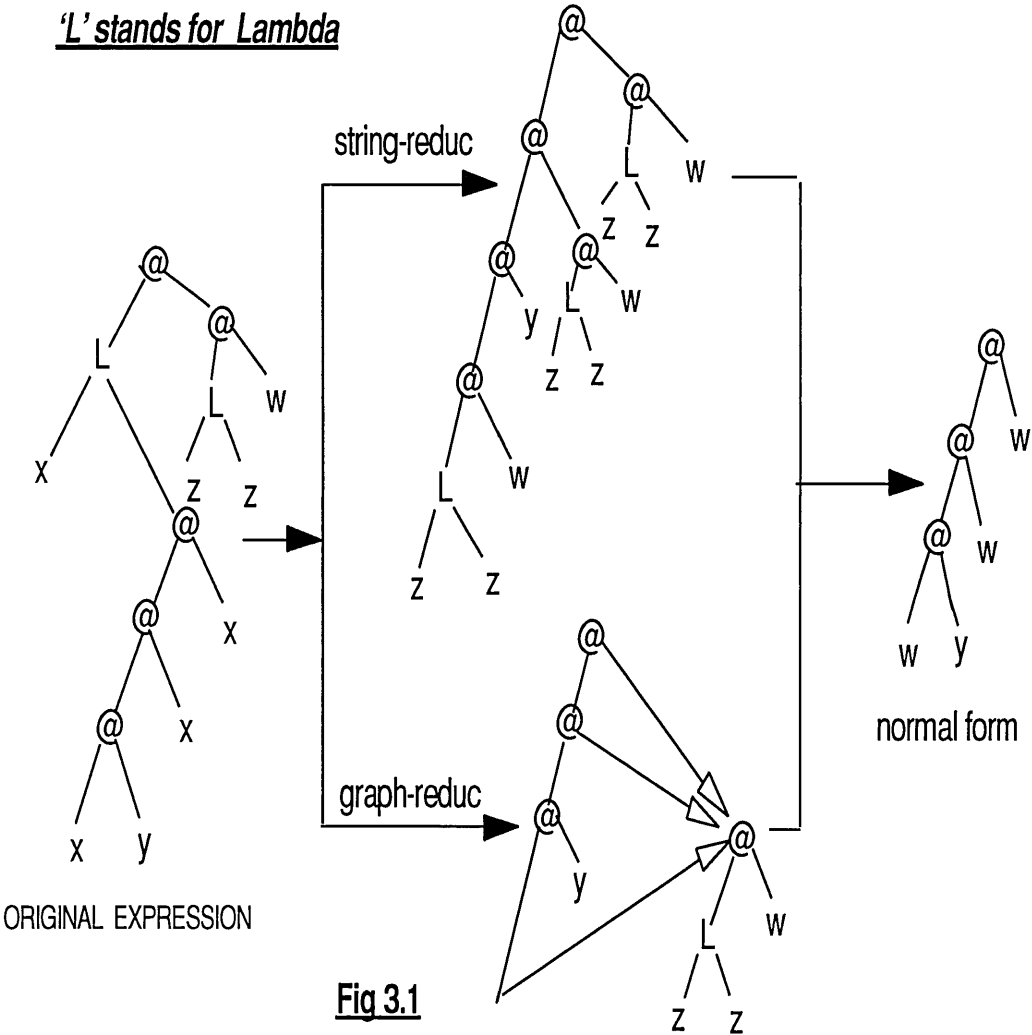
3.1 Graphical representation of λ -expressions

It was stated in the previous chapter that, to reduce the input expression (λ -expression) to a normal form, the substitution of a bound variable with an argument is achieved by replacing the variable reference by a pointer to the argument graph. This means that in the event that the function body contains references to the bound variable, multiple pointers to a single shared copy of the argument graph will be established. Consequently that the same copy of a sub-program can be shared

among several parts of a larger expression. For example consider a λ -expression:

$$(\lambda x. x y x x)((\lambda z. z) w) \xrightarrow{\beta} (((((\lambda z. z) w) y) ((\lambda z. z) w)) ((\lambda z. z) w))$$

Its construction is given on figure 3.1, the lower part of fig 3.1 represents the graph representation.



There are other systems as well. The alternative, of copying the argument wherever it is used, is called string reduction or tree reduction.

3.1.1 String reduction

String reduction is like a graph reduction except the value of an expression is not shared among multiple references. In graph reduction the two occurrences of the same identifier refer to the same expression rather than by copying that expression out twice as in string reduction. Figure 3.1 (upper part) makes this clear. The expression contains three occurrences of the identifier 'x', so three copies of the identifier 'x' will be made in string reduction.

A simple example can be given to explain the above larger λ -expression :

graph reduction:

$$\begin{aligned} (\lambda x. * x x)(+ 4 5) &\xrightarrow{\beta} * x x \text{ where } x \text{ is } (+ 4 5) \\ &\xrightarrow{\beta} * 9 x \text{ where } x \text{ is } 9 \\ &\longrightarrow * 9 9 \\ &\longrightarrow 81 \end{aligned}$$

string reduction:

$$\begin{aligned} (\lambda x. * x x)(+ 4 5) &\xrightarrow{\beta} * x x \text{ where } x \text{ is } (+ 4 5) \\ &\xrightarrow{\beta} * (+ 4 5) x \text{ where } x \text{ is } (+ 4 5) \\ &\longrightarrow * 9 (+ 4 5) \\ &\longrightarrow * 9 9 \\ &\longrightarrow 81 \end{aligned}$$

This means after evaluating the first x , the next time x is required the reduced form of x (previous x) will be delivered immediately, and so in graph reduction the argument $(+ 4 5)$ is been evaluated only once.

String reduction has another disadvantage for conditional expressions such as:

if $w = 1$ then $g(w)$ else $h(w)$

where three copies of w are made even though the argument w will be used by only one function, so one copy of w will be thrown away.

The string reduction some times leads to unnecessary copying of the arguments, and it is normally considered prohibitively expensive [43]. However it has been used (in Mago's parallel reduction machine) as the basis for an architecture allowing massive parallelism [61]).

Replacing one graph by another with the same value, means it is no longer required in the evaluation, and so the space occupied can be reclaimed.

As a result, logically adjacent parts of an expression will not necessarily occupy adjacent storage locations. The segments of storage used and then released dynamically during the reduction process may become widely scattered all over the storage space.

As execution proceeds more and more space will be allocated in the memory. This allocation will finally fill up the memory space, so we need to perform garbage collection to free up some spaces.

3.1.2 Graph reduction on Rekursiv

The Rekursiv's storage management system has features which may well support graph reduction.

The garbage collection technique of the Rekursiv provides

- garbage collection in hardware, that is, there is hardware for generating certain condition codes to assist the process of garbage collection.
- memory management based on arbitrary-sized objects

- performing all necessary checking in parallel with other processing to manipulate objects in a secure persistent environment. The Rekursiv has a number of registers dedicated to specific functions.

So by giving the responsibility of recycling of an unused piece of fragmented store to the hardware, the Rekursiv is ideal for graph reduction.

3.2 Combinator reduction scheme

The early implementations of functional languages either reduced λ -expressions directly, or translated them into a program for an SECD machine. Both of these approaches were rather slow as implementations of normal order reduction because they had to do a lot of copying.

Turner produced an implementation that was superior to these reducers by translating λ -expressions into expressions containing only combinators. The idea is based on the well-known fact that all of the variables in a λ -expression can be transformed into an equivalent form that does not include any λ .

3.3 λ -calculus Vs combinator reduction

A combinator is rather like a λ -abstraction but provides three advantages over λ -abstraction.

- all the arguments to a combinator are dealt with at once (discussed below)
- there are no free variables in the combinator's defining body
- a combinator body is an application

These differences indicate that combinator reduction may be performed faster than the reduction of arbitrary λ -expression. As their bodies are constant forms, combinators can be compiled into an intermediate code suitable for speeding the reduction process. In fact only two combinators S and K (see below) are needed to translate a λ -expression to an applicative expression. For example

$$\lambda x. \lambda y. a \ x \ (b \ y) = S(S \ (K \ S)(S \ (K \ K) \ a)) \ (K \ b)$$

The important property is that any λ -expression —and hence functional program— can be transformed into an expression consisting solely of applications of these combinators. These combinators (S K) are devised to prevent unnecessary copying of arguments.

3.4 Turner's reduction scheme

The idea of implementing a functional language by translating expressions into combinatory form and performing reduction on the result is due to Turner [65].

Turner's combinatory scheme uses the following operators (combinators):-

$$S \ f \ g \ x = f \ x \ (g \ x)$$

$$K \ x \ y = x$$

$$I \ x = x$$

where SKI can be written in terms of λ -calculus as

$$S = \lambda f. \lambda g. \lambda x. f \ x \ (g \ x)$$

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

$$I = \lambda x. x \text{ Only S and K are necessary, as } I = SKK.$$

As an expression undergoes reduction, the intermediate expression generated tends not to display the combinatorial growth in size that is typical when using only the SK and I combinators. This has been referred to as a 'self-optimizing' property of combinator reduction, and seems to occur as the set of judiciously chosen dependent combinators reaches a critical size [61].

The abstraction operations are

```
[x] (E1 E2) => S ([x] E1) ([x] E2)
[x] x      => I
[x] y      => K y
```

where: y is a constant or a variable other than x; E1 & E2 are expressions;
S K & I are three combinators each having a strictly limited power of beta conversion.

Consider an example of the factorial function.

```
fact = λx. if x = 0 then 1 else x * fact (x - 1)
      ⇒ cond (= x 0) 1 (times x (fact (- x 1)))
```

abstracting [x] by representing as:

```
[x](cond (= x 0) 1 (times x (fact(- x 1))))
```

results in the following SKI expression:

```
S (S ( S (K Cond) (S (S (K =) I) (K 0))) (K 1)) (S (S (K Times) I) (S
(K fact) (S (S (K -) I) (K 1)))))
```

The tree representation is:-

It is remarkable that the SKI calculus is computationally complete, that is these three simple symbol-manipulation operations are sufficient to implement any operation by translating lambda calculus formulae into formulae in the SKI calculus. Since functional languages can be compiled into lambda-calculus and lambda-calculus can be compiled into SKI, It follows that any functional language can be implemented on a computer providing just the SKI operations. However as Turner points out, an implementation relying solely on S, K and I is hopelessly inefficient. Because a formula of the SKI contains no bound identifier, its reduction can be implemented as simple data structure manipulations. Further the reduction can be applied in any order [52].

3.4.1 Turner's optimization

Fixing the set of combinators in advance allows the design of specific hardware for combinator evaluation [60].

Using SKI the translation produces expression which are larger than the original by more than the square of the number of terms. With the few more combinators and reduction rules the resulting expression are more compact. Additional combinators representing primitive operations, such as arithmetic operators may be used for efficient execution. Other nonprimitive combinators like B and C can also be added, so as to reduce the size of the resultant combinator expression.

The combinators B and C can be defined by :-

$$B \ f \ g \ x \ = \ f \ (\ g \ x \)$$

$$C \ f \ g \ x \ = \ f \ x \ g$$

$$S \ (K \ E1) \ (K \ E2) \ ==> \ K \ (E1 \ E2)$$

$$S \ (K \ E1) \ I \ ==> \ E1$$

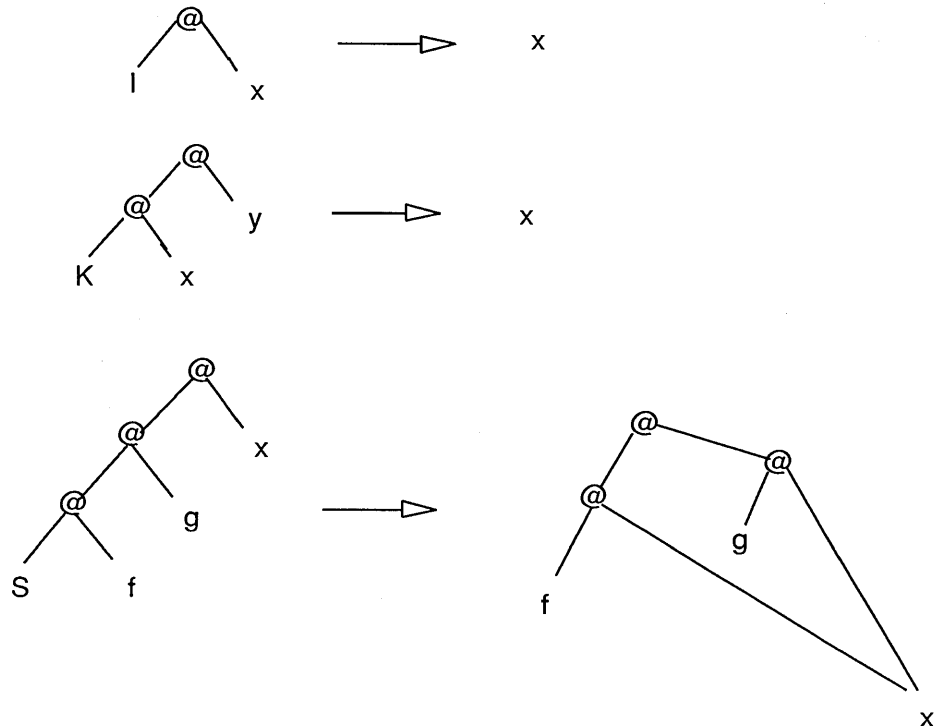
$$\begin{aligned}
 S (K E1) E2 &\Rightarrow B E1 E2 \\
 &\quad \text{(if no earlier rule applies)} \\
 S E1 (K E2) &\Rightarrow C E1 E2 \\
 &\quad \text{(if no earlier rule applies)}
 \end{aligned}$$

B & C optimize the combinator code of 'fact' to :

$$S(C(B \text{ cond}(C = 0))1)(S \text{ times}(B \text{ fact } (C - 1)))$$

With these optimisations graph reduction becomes a practical implementation technique, with the important advantage of full laziness [37].

Graphical representation of these combinators are given in fig 3.4



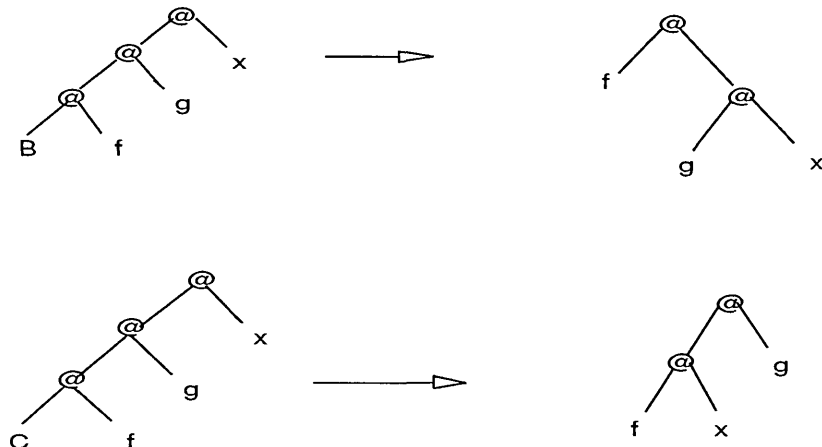


Fig 3.4

Graph transformation rules for SKIBC combinators

Similarly the reduction rules for prime combinators like S' , B' and C' can also be defined (see chapter-4.3.1).

3.4.2 Representation of recursion

In mathematical functions there is no concept of control flow since functions are defined declaratively. Instead recursion and conditional expressions are used.

A good example is the factorial function which involves both conditional expressions and recursion (Wilson, pp.150,[73]). It is written in Pascal. The same program may be represented in mathematical function as:

```
def fact = lam x. if x = 0 then 1 else x*fact(x-1)
```

The application of function 'fact' to an argument '4', will compile to a tree whose left subtree is the structure of the 'fact' and whose right subtree is the argument 4.

The definition of fact can be used until an internal reference to fact occurs again. The function name 'fact' is included in the symbol table. When function name fact reappears in the expression, a definition of fact with internal references to fact is

replaced, that is, wherever the function definition fact is found in the expression, the reduction process will retrieve the tree associated with the name in the symbol table and incorporate it. This process goes on until the argument supplied to fact becomes zero and the recursion stops.

The lazy approach is ideal for recursive function evaluation [74].

A complete example of a recursive function is detailed in chapter-4.3.1. In this example the recursion representation involves SKI and optimized combinators B, C, S', B', and C'.

3.5 Supercombinators

It was mentioned earlier in the SK combinator section (section 3.4.1) that by adding more combinator efficiency is improved, which results in a reduction of the size of the graphs and in the number of reductions needed for program execution.

This SK combinator-based compilation strategy provides a fixed set of combinators into which programs are translated. The evaluator is also designed to reduce a graph of combinators from this fixed set. S K combinators are sufficient for representing arbitrary functions [30]. Theoretically any computable function can be expressed as an S K combinatory expression [33]. Using a small set of combinators as in Turner's system means that a program will be executed in very small steps. This representation, however, is rather lengthy and hence inefficient.

The supercombinator (λ -lifting) approach has the advantage over the fixed combinator approach in that each supercombinator produced is almost certain to be larger than the fixed combinators, so that fewer applications are required to do the same amount of work, ie the grain size of each combinator is increased.

Hughes's supercombinator approach [37] generates efficient representation using

program specific combinators. They are generated directly from the λ -abstractions involved. This technique treats all free variables as arguments to additional λ -abstractions. Constructing an instance of a λ -body while substituting for the formal parameter (ie the variable bound by the λ) is the fundamental operation of this type of implementation.

Applying a lambda-abstraction to an argument involves constructing an instance of its body, and substituting the argument for occurrences of the formal parameter. For example:

$$(lam\ x.\ x + 1)\ 3 \implies +\ 3\ 1 \implies 4$$

The above expression may be represented by one supercombinator. The reduction efforts may be broken down into two steps:

- substitute the argument (in this case '3') for occurrences of the formal parameter (x in this case)
- perform addition

In the case of nested λ -abstraction the effect of this is to generate a different set of supercombinators from each λ -abstraction.

Consider the lambda-abstraction

$$lam\ x.\ lam\ y.\ x + y$$

x occurs free in the body of $lam\ y$ abstraction (y being the bound variable). By providing two arguments (one for x and one for y) this above expression may be reduced to two supercombinators.

This technique generates 'tailor made' combinators from a program to be the best for that particular program. Hughe's tailor-made combinators can be made as large as possible.

As we mentioned before, combinatory logic can be used to remove all the bound variables from a functional program, producing a constant application form (CAF), in which there are no variables. The same idea applies here, we transform the program by translating each lambda-abstraction into CAF using supercombinators.

Each supercombinator is a pure function and so can be viewed as a constant until all expressions are in CAF.

3.5.1 Implementation

In order to transform an arbitrary lambda-term into supercombinators, it is necessary to review the concept of free variable/expression (see chapter 2). For example

$\lambda y. y$ is a supercombinator(S.C.) (no free variable)

$\lambda x. x + 1$ is a S.C. (no free variable)

$\lambda x. x (\lambda y. (y * y))$ is a S.C. (no free variable)

$\lambda x. (x + y)$ is not S.C. because y occurs free.

$\lambda x. x (\lambda y. x * x)$ is not a S.C. in inner λy abstraction x occurs free.

To illustrate this mechanism consider the following function:-

$def\ a = lam\ x. (lam\ y. (x + (lam\ x. (x - 3))\ y))\ 5$ —————(1)

The expression may have many λ abstractions which are not supercombinators, ('def' in the above expression, is an additional facility for naming expressions as in Alcal).

In the above example select a λ -abstraction which has no inner λ -abstraction in its body, which is $lam\ x. x - 3$; This is a supercombinator, since no variable occurs free in its body.

Give that supercombinator an arbitrary name say $N1$. The definition of $N1$ is given as

$$N1 := x - 3 \implies - x 3;$$

The reduction of supercombinator $N1$ can be defined as: ‘substitute the argument for x and then subtract 3 from it.

The original expression (1) has now been transformed into

$$\text{lam } x. (\text{lam } y. (x + N1 y)) 5$$

The inner *lam* term is now $\text{lam } y. (x + N1 y)$. Note that x occurs free. This can be rewritten as:

$$(\text{lam } x. \text{lam } y. (x + N1 y)) x$$

$\text{lam } x. x$ being an identity function which removes the free variable from the above expression. The inner bracketed term is a supercombinator, which we call $N2$.

$$N2 := x + N1 y \implies + x N1 y$$

The original expression (1) has been further reduced to:

$$\text{lam } x. N2 x 5$$

The reduced form is itself a supercombinator which is the Identity combinator, so our final term is a CAF ie $N2$ applied to 5:

$$N2 5$$

This example shows that the original expression (1) generates only two supercombinators, namely $N1$ and $N2$. The definition of $N2$ indicates that it takes two arguments, and applying $(N2 5)$ to 6 will achieve the result.

The grain size of the supercombinators $N1$ and $N2$ is larger than that of SK

The grain size of the supercombinators *N1* and *N2* is larger than that of SK combinators. The resulting code produced by Alcal (for the above example) is:

```
((((C (((C' B) +) (( C -) 3))) 5)) 6
```

Alcal has to carry out 9 reductions to evaluate the expression, while the algorithm of lambda-lifting requires only two.

3.5.2 Parameter ordering and Redundant parameters

In the case where several free variables are to be taken out from a lambda-abstraction as extra parameters, the free variables should be ordered with those bound at inner levels coming last in the parameter list of the supercombinator. For example in our original example

```
lam x. ( lam y. (x+(lam x. (x-3))y ))5
          |<-level-1-->|
          |<-----level-2----->|
          |<-----level-3----->|
```

Parameter ordering helps in connection with efficiency. Making a parameter redundant causes a supercombinator to be redundant. Eliminating the redundant parameters is a simple optimization of lambda-lifting.

An example is given to explain redundant parameters and parameter ordering.

3.5.2.1 Redundant parameter

Consider an example which involves two variables:

```
lam x. lam y. x + (y * y) -----(2)
```

Performing lambda-lifting, the inner lambda-abstraction is $\text{lam } y. x + (y * y)$ and x is free.

The first supercombinator is generated as: ‘ $\text{lam } x. (\text{lam } y. x + (y * y)) \ x$ ’ with the name $N1$ and parameter list $x \ y$ as $(N1 \ x \ y)$ where the definition of $N1$ is

$$N1 := (x + (y * y)).$$

Substituting back $N1$ into the top example (2) gives

$$\text{lam } x. N1 \ x$$

Here we have only a lambda- x abstraction and no free variables so it is a supercombinator. This may be coded as $N2$ with parameter x and its definition is:

$$N2 := N1 \ x.$$

Now the two supercombinators with their parameters are

$$N1 \ x \ y = N2 \ x$$

Here $(N2 \ x)$ is redundant. The supercombinator $(N2 \ x)$ may be replaced by $(N1 \ x)$ wherever $(N2 \ x)$ occurs. It is clear that we require only one supercombinator $(N1 \ x \ y)$ with the definition of $N1 := x + (y * y)$.

3.5.2.2 Parameter ordering

The same above example may be extended (adding one more variable) for an explanation of parameter ordering.

$$\text{lam } x. \text{lam } z. y + (x * z) \text{ ————— (3)}$$

The inner lambda term is $\text{lam } z. y + (x * z)$, where x and y are both free, and the order in which we take them out makes a difference to the resulting combinators.

more free the variable. ie x is more free than y and y is more free than z . Simply it may be defined to be the order of the free variables with those bound at inner levels coming last in the parameter list of that supercombinator: (2) with 'z' gives -

$$\text{lam } z. \text{ lam } x. (\text{lam } y. x + (y * z)) z x \implies (N1 z x) z x$$

The first combinator is $(N1 z x)$ with definition $N1 := (x + (y * z))$. Substituting back to (2) with 'z' yields

$$\text{lam } x. (N1 z x) z x$$

Note: ' $(N1 z x) z x$ ' is a supercombinator with parameter list $z x$ and that supercombinator is applied first to an argument z and then to an argument x . It is clear that in the above expression z is free.

The next supercombinator may be generated as

$$(\text{lam } z. \text{ lam } x. (N1 z x) z x) z \implies (N2 z) z$$

Two supercombinators with their parameters are

$$N1 z x = N2 z$$

$N2 z$ becomes redundant leaving only one supercombinator to evaluate. A different ordering of parameters would result in both supercombinators having to be evaluated. This is a small optimization and this redundancy makes the evaluation lazier.

3.5.3 Lifting with recursion

No special note need be taken of recursion, it can be dealt with in much the same way as in SKI-implementation. In the presence of a supercombinator, the recursive call to the supercombinator is made directly to that supercombinator. It is easier

to illustrate this with an example of the factorial function.

Assuming 'N1' is the supercombinator of the factorial function with the definition of:

```
(Cond = x 0) 1 (* x (fact (- x 1))).
```

Its implementation is similar to SKI-implementation except that the recursive call to fact can directly be replaced by the supercombinator N1. This algorithm for recursion is fully lazy.

3.5.4 Conditional handling and functional application

The conditional handling and functional application are similar to SKI-implementation.

3.5.5 Identifying lambda-abstractions

Lambda-abstractions of the form of $\lambda x. E$ are represented by a special type of node, called the lambda-node (see below), with the bound variable associated with the left branch and the right sub tree representing the expression body.

```
    lam
   /  \
  /    \
 /      \
char node
```

The body of the expression may include the occurrences of bound variables of other lambda-abstractions as free variables, (in our original example (1) see the generation of supercombinator N2, where x is free in that body expression but it is bound to

the outer lambda-abstraction(lambda x)). Presently the code is being generated directly from the source program.

3.5.6 Identifying free variables

The operator for finding free variables may be given as (Hope):-

```
data node = apply (node X node) ++
            lambda (char X node) ++
            numb (num) ++
            var (char) ++
            op (char)

dec ff :: list char X node ---> list char;
--- ff (x,apply (n1,n2)) ---> ff (x,n1) <> ff (x,n2)
--- ff (x,lambda (v,n)) ---> ff (v :: x,n)
--- ff (x,numb (n)) ---> nil
--- ff (x,var (v)) ---> if member (v,x)
                        then nil
                        else {[v]}
```

Hope syntax is used here because the use of data type 'node' clarifies the function definition.

3.5.7 The Evaluator

With the original example (1) described in the implementation section:

```
def a = lam x. (lam y. (x + (lam x. (x-3)) y)) 5
```

It generates two supercombinators, N1 and N2. For evaluating N2 at argument say 6, the evaluation proceeds as:

```

(N2 5) 6 => (((+ x) N1) 5) 6
=> (((+ x) (((- x) 3) y)) 5) 6
=> (((+ x) (((- x) 3) 5)) 6
=> ((+ x) ((- 5) 3)) 6
=> ((+ x) 2) 6
=> ((+ 6) 2)
=> 8

```

Initially =>

$$\begin{array}{c} @ \\ / \backslash \\ @ \quad 6 \\ / \backslash \\ N2 \quad 5 \end{array}
 \text{ where } N2 => \begin{array}{c} @ \\ / \backslash \\ @ \quad @ \\ / \backslash \quad / \backslash \\ + \quad x \quad N1 \quad y \end{array}
 \quad N1 => \begin{array}{c} @ \\ / \backslash \\ @ \quad 3 \\ / \backslash \\ - \quad x \end{array}$$

Finally =>

$$\begin{array}{c} @ \\ / \backslash \\ @ \quad 6 \\ / \backslash \\ @ \quad 5 \\ / \backslash \\ @ \quad @ \\ / \backslash \quad / \backslash \\ + \quad x \quad @ \quad y \\ / \backslash \\ @ \quad 3 \\ / \backslash \\ - \quad x \end{array}$$

As we have seen above the parameter ordering yields an efficient result, but in the case of CAFs no benefit is obtained.

In the case of $\text{lam } x.x+1$ the term $(+ 1)$ is a CAF with primitive combinator Plus. $(+ 1)$ being shared for each application of lambda-abstraction. In cases where the

CAF are redexes like $(1+(2+3))$ this will save the repeated evaluation of $(1+(2+3))$ because it is a supercombinator.

3.5.8 Advantages over SKI-combinators

- The program dependent supercombinators generated are larger than the fixed combinators, require fewer application to do the same amount of work, ie the grain size (execution steps) of each supercombinator is increased. Even though lambda-lifting is quite elegant (more efficient than SKI), because of its larger grain size, it is not fully lazy and so not optimal. Optimization may be carried out by adopting the concept of a maximal free expression (m.f.e). In lambda-lifting, we abstract out the free variables as extra parameters while in m.f.e the entire free sub-expression is abstracted out as an extra parameter.

One more possible way to achieve the optimization is to define each supercombinator as a local variable until the final supercombinator is generated and then perform the evaluation. In this way there will be one only supercombinator for a program and it will be fully lazy.

- Supercombinators produces less resulting code than SK combinator reduction, so it is generally faster than SK combinator implementation.
- supercombinators place less load on memory management (garbage collection).
- Generating supercombinators is more complex than SK combinators. In the supercombinator technique we add extra formal parameters in response to free variables appearing in the expression by repeatedly replacing the innermost λ -abstraction using a suitable abstraction algorithm.

Supercombinators have formed the basis for high performance implementations of functional languages.

The λ -lifting technique appears currently to be the preferred approach to the efficient implementation of functional languages [38, 43].

Chapter 4

An Implementation of Combinator Reduction

This chapter describes the implementation scheme of the combinatory logic functions as the machine code that a purely applicative language (Alcal) is translated into. Functional programs are translated into ‘active’ graphs. The execution of these active graphs is described. The chapter begins by giving a description of the language, its handling of various features like recursion, higher order functions etc. Then the abstraction algorithm with various optimizations (reducing code size) is described and finally the evaluation strategy of Alcal is handled.

4.1 The language; syntax and example

Alcal (A λ -calculus language) is an applicative language. It is a direct extension of the λ -notation. Its syntax is set out in Appendix-A. An interpreter for Alcal was developed using the recursive descent method [16] to arrive at an implementation almost directly from the extended BNF specification [56].

Alcal is designed to provide facilities for functional programming. The programmer need not be concerned with variables. Program execution proceeds by function application.

For example, a function definition may be written as:

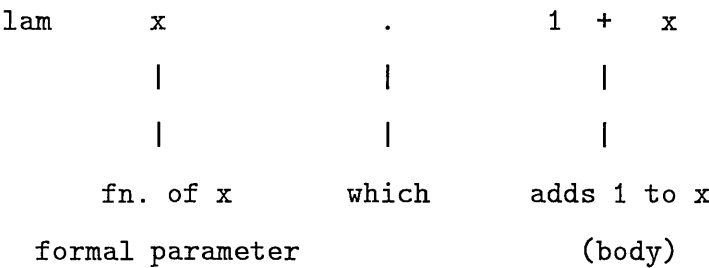
```
def f = lam x. 1 + x;
```

This defines a function f . The lam (λ) notation on the right hand side of the definition declares that f takes an argument.

This is similar to the Pascal function:

```
function f(x:integer):integer;
begin
  f := 1+x;
end;
```

The variable x (ie the variable bound by the λ) is called the formal parameter.



A λ -abstraction always consists of these four parts, that is, the keyword lam, the formal parameter (x in this case), the '.' and the body ($1+x$ in this case). Thus to get the result of applying the λ -abstraction (lam x . $1+x$) to the argument 5 is

represented as 'f 5' (which means f applied to an argument 5). The mechanics of the above application is same for all function evaluation in Alcal.

4.1.1 Naming the function

As shown in the above example the general definitions appear on the left hand side of the expression. For example:

```
def add = lam x.lam y. x+y;  
add 2 3 -> 5
```

```
def twice = lam x. 2*x;  
twice 4 -> 8
```

```
def length = lam x. if x = nil then 0 else 1+length(tl x);  
length [1,2,3] -> 3
```

```
def map = lam f.lam l.if l = nil then nil else f(hd l):: map f(tl l);  
map twice [1,2,3] -> [2,4,6]
```

In this way the definition 'def' allows us to make a simple check on recursion as well. A symbol table is used to associate named functions with their corresponding graphs, if the same name reappears on the right hand side, that is in the body (see the function definitions 'length' and 'map' in the above given example), we retrieve the graph associated with that name in the symbol table and incorporate it.

4.1.2 Use of brackets

Alcal supports the omission of redundant brackets to avoid cluttering up the expression. According to the conventions that function application associates to the left, we can write a complex example as: (factorial function)

```
def fact = lam x. if x = 0 then 1 else (x * (fact (x - 1)))
```

the expression is fully bracketed and unambiguous (specially the last part of the expression). We may omit redundant brackets to make the expression easier to read as:

```
def fact = lam x. if x = 0 the 1 else x * fact(x-1)
```

Now suppose that we required to evaluate this expression then the function fact applied to 4 may be represented as: 'fact 4'. Extra brackets may, of course be inserted freely without changing the meaning of the expression. For example (fact 4) is same as 'fact 4'.

4.1.3 Microsyntax

The micro syntax of Alcal reduces to just a few terminal classes, which describes the syntax of integers or any upper or lower case alphabetic character (identifier).

A class, Scanner was implemented in Lingo to scan the input source stream. It includes a class method for instantiation which takes as parameter collections of keywords for example:

```
'def', 'if', 'then', 'else', 'lam', 'div', 'and', 'or',  
'not', 'exit', 'hd', 'tl', 'true', 'false', 'nil' 'mod'
```

punctuation characters for example:

[. ()] ; % + - /

cryptic character like:

= < > * :

and also a file descriptor for the source text. By reading the source program one character at a time, the source program is carved into a sequence of atomic units called tokens. Each token represents a sequence of characters that can be treated as a single entity or identifiers or keywords etc.

Keywords like 'def, if, then' etc could be considered to be identifiers, punctuation characters are those characters that can not prefix other characters, for example ',', '.', ') etc. Characters like '<' or '=' may have prefix characters like '! =' or '<=' so they are grouped as cryptic characters.

The instance methods of class Scanner include 'getToken' which returns the current token and advances through the source stream. There are also methods 'theInteger' and 'theIdentifier' which return the actual values found in the source stream for 'Integer' and 'Identifier' respectively. The next paragraph describes recognizing variables and identifiers.

4.1.4 Distinguishing identifier and variable

To distinguish between an identifier and a variable in the input stream, identifiers are entered into a symbol table and then it is possible to determine whether they are indeed variables or not.

To do this, a class, MCnew is defined as a specialisation of Scanner. It has an additional instance variable, 'symbolTable' to hold the variable and access methods

to place and look up variables in this table. The inherited `getToken` method calls the superclass `getToken` and then inspects the returned token. If it is an identifier then the `symbolTable` is searched (it is performed by method `'self item'` of class `MCnew`). If `symbolTable` includes the identifier then `'Variable'` is returned else `Identifier` is returned.

In addition there is a method `'mustBe:'` which is also included in class `MCnew`, which checks that the token it is passed is the same as that which is required, then get next token otherwise report the syntax error.

4.1.5 Syntax error

The cause of error should give the user clear information, for correcting the errors. For example consider the following error

```
if      then y  else  z
```

Alcal gives an error as: `'found then when expecting Identifier'`. For this reason an instance variable `'recovering'` is also introduced in the class `MCnew`.

It is not possible to recover by merely scanning till a statement terminator is found, since at the time of the error, the thread of execution will in general be at some deeply nested point due to the dependence on recursion. Alcal handles this by using Davie's [16] elegant algorithm for error recovery in such a situation, which merely adds a few lines to the `mustBe:` method. The Lingo [31] provides an exception facility. Initially `'recovering'` is set to boolean false, and is set true when an error has been detected. When an error is detected `'recovering'` turns true and then the rest of the input is skipped until the terminator symbol `(;)` is found and then a syntax error is reported. This mechanism can be used to allow return to the top level of syntax analysis on encountering the first syntax error.

4.2 Representation of function definitions

Some features of Alcal that the implementation must support may be summarised as follows:

4.2.1 Occurences of formal parameters

The formal parameter may occur several times in the body. e.g.

```
def f    = lam x. 1+x+x
      f 5 =      1+5+5
      =      11
```

There may be no occurences of the formal parameter at all in the body. e.g.

```
def f    = lam x. 1
      f 5 =      1
```

here in this case the argument is discarded unused.

4.2.2 Multi parameters

λ -expressions, like other function definitions, may be applied to an appropriate number of arguments (one at a time). There may be more λ -abstractions in the body. e.g.

```
def f    = lam x. (lam y. x+y)
      f 5 6 = (lam y. 5+y) 6
      = 5+6
      = 11
```

here the λx abstraction returned a function (the λy . abstraction) as its result, which when applied to 6 gives the final result. We start by substituting 5 for x through the body to end up with the application of another abstraction to an argument expression, this time substituting 6 for y.

4.2.3 Higher-order functions

Functions may be used as arguments. Complex functions may be defined in terms of other functions. e.g.

```
def f = lam x. 1+x;
def g = lam x. x-1;
the function h may be defined as
def h = lam x. f(g(x))
```

```
h 5 = f(g 5);
    = f(4);
    = 5;
```

4.2.4 Booleans and the conditions

On encountering conditional operators, the condition should be evaluated to determine which subtree to preserve (It is already mentioned in chapter-3 that a functional program is represented as a tree or more generally as a graph).

```
if b = 0 then p else q ----> (((Cond ((= b) 0)) p) q)
```

It is simply, if b is zero then the result of the whole term

```
(( ( Cond ((=b) 0)) p) q)
```

will be p otherwise q.

The examples of other truth functions like Not, And, Or are similar. The truth functions Cond, And are designed to be lazily evaluated in the Rekursiv. That is, they will evaluate their second operand only if the first operand does not decide the issue.

4.2.5 List notations

In Alcal a list can be on nil-form ie ([]) or cons-form ie (::). The operator ::, which is called cons creates a new list from an element and a list. For example

```
(1 :: (2 :: (3 :: nil)))
```

represents a list of integers with three elements. It was mentioned earlier that in Alcal we may omit redundant parentheses for clarity. The above list can be represented as:

```
(1::2::3::nil)
```

The same above list can also be represented as:

```
[1,2,3]
```

and the both forms may be combined as:

```
5 :: [3,6,9]      ---> [5,3,6,9]
[4] :: [5] :: nil ---> [[4],[5]]
[[4],[5]]         ---> [[4],[5]]
[1,2]::[3]::nil   ---> [[1,2],[3]]
[[1]::[2]::nil]::[[3]::[4]::nil]::nil ---> [[[[1],[2]]],[[[3],[4]]]]
[true,false,true] ---> [Boolean true,Boolean false,Boolean true]
```

4.2.5.1 List operators

In Alcal the fundamental list operators Head, Tail and Cons are represented as: `hd`, `tl` and `::`. The first element of a list is selected with `hd` and the rest of the elements are selected by `tl`. The operator `::` is already described in a previous section.

```
hd [1,2,3] ----> 1
```

```
hd(1::2::3::nil) ----> 1
```

```
hd[[5,12],[10,15],[15,23]] ----> [5,12]
```

(ie head of list of lists)

```
tl [1,2,3] ----> [2,3]
```

```
tl(1::2::3::nil) ----> [2,3]
```

```
tl[[5,12],[10,15],[15,23]] ----> [[10,15],[15,23]]
```

(ie tail of list of lists)

‘nil’ may also be defined as:

```
def isnil = lam x. if x = nil then 0 else 1;
```

```
isnil nil ----> 0
```

```
isnil 0 ----> 1
```

```
isnil 1 ----> 1
```

Similarly ‘and’, ‘or’ and ‘not’ can also be defined.

4.2.5.2 Standard list handling functions

After defining the Cons operator in Alcal, the implementation of most standard functions (including recursive ones) may easily be hand coded (see Appendix-C). For example:

```

def append = lam a.lam x.if x = nil then a::nil else hd x::append a(tl x);
def concat = lam x. lam y. if x = nil then y else hd x :: concat(tl x)y;
def twice  = lam x. 2*x;
def map    = lam f.lam l.if l = nil then nil else f (hd l):: map f(tl l);
def maptw  = map twice;
def fold   = lam f. lam i.lam l.if l = nil then i
    else f (hd l) (fold f i (tl l));
def sumlist = lam x.if x = nil then 0 else hd x + sumlist (tl x);

```

```

append 1 [2,3,4]      -> [2,3,4,1]
concat [1,2,3][4,5]   -> [1,2,3,4,5]
map twice [1,2,3]     -> [2,4,6]
maptw [1,2,3]         -> [2,4,6]
fold append [1][2,3]  -> [1,1,1]
fold append [][1,2,3] -> nil
    sumlist[1,3,8]     -> 12

```

4.2.6 Recursion handling

Alcal also handle the recursive functions as described earlier in this section. e.g.

$$\text{def fact} = \text{lam } x. \text{if } x = 0 \text{ then } 1 \text{ else } x * \text{fact}(x - 1);$$

Wherever the definition ‘fact’ reappears in the expression, an internal reference to fact (a tree in the symbol table associated with this dummy name) is replaced, until the condition is satisfied and recursion stops.

4.2.7 Intermediate code in prefix form

Before describing the Abstraction representation some examples may be given in their respective prefix curried forms generated by Alcal: (see Appendix-C as well)

```
def fact = lam x.if x = 0 then 1 else x*fact(x-1);
Alcal -----> (((cond ((= x) 0)) 1) ((* x) (fact ((- x) 1))))
```

```
def fib = lam n.if n < 2 then 1 else fib(n-1)+fib(n-2);
Alcal -----> (((cond ((< n) 2)) 1) ((+ (fib ((- n) 1)))
                                         (fib ((- n) 2)))))
```

```
def concat = lam x. lam y.if x=nil then y else hd x :: concat(tl x)y;
Alcal -----> (((cond ((= x) nil)) y) ((:: (hd x))
                                         ((concat (tl x)) y)))
```

```
def ack = lam x. lam n.
    if x = 0
    then n+1
    else if n = 0
        then ack(x-1) 1
        else ack(x-1)(ack x(n-1));
Alcal ---> (((cond ((= x) 0)) ((+ n) 1)) (((cond ((= n) 0))
                                         ((ack ((- x) 1)) 1))((ack ((- x) 1))((ack x) ((- n) 1))))))
```

Alcal provides an auxiliary function called **load** < *string* > as well. e.g.

```
load "benchall" ;
```

The function load scans all the function definitions in the file 'benchall' into the executable code with their respective prefix form until an 'exit' is encountered,

which terminates it. This facility enables us to load the Alcal with more functions in one go.

4.3 Abstract representation

Turner [65] outlines a method of translating functional expressions into expressions involving the combinators S, K, I. The method is known as bracket abstraction and is usually described by the following rules: (see also chapter:3.4)

$$[x] \ x \quad = \ I \quad \dots(1)$$

$$[x] \ y \quad = \ K \ y, \text{ if } y \not\rightarrow x \quad \dots(2)$$

$$[x] \ M \ N \ = \ S([x]M)([x]N) \quad \dots(3)$$

Here x and y are variables and M, N are arbitrary terms which may or may not contain x . The notation ' $[x]M$ ' denotes the result of abstracting the variable ' x ' from the function M . An example should make this clear, consider the Alcal function definition:

```
def f = lam x. 1+x
```

Turner shows how to translate such an expression (in curried form) into a combinator form (in which there are no bound variables) using the above three bound variable abstraction rules. In the following, abstracting ' x ' from the expression ' $+ 1 \ x$ ' is denoted as ' $[x] ((+ 1) \ x)$ '.

$$\begin{aligned} [x] \ ((+ 1) \ x) &\longrightarrow S([x](+ 1))([x]x) && \text{by using (3)} \\ &\longrightarrow S(S([x] +)([x]1)))I && \text{by (3) and (1)} \\ &\longrightarrow S(S((K +)(K 1)))I && \text{by (1) and (2)} \end{aligned}$$

If the body of the expression is an application then apply the S-transformation rule (3), otherwise the body must be a variable or a constant, so apply rule (2) or (1) for K or I transformation as appropriate.

We transform the expression into an expression involving only S, K, I and constant. By transforming the lambda-abstraction we ensure that the body of that lambda-abstraction contains no lambdas. The resulting SKI expression can then be ‘executed’. The definition of each of these combinators and their graphical representation as per their reduction rules has been given in previous chapter. To summarize the transformation rules and the reduction rules for the combinators S, K and I are given as follows:

S-transformation: $\lambda x. M N = S(\lambda x. M)(\lambda x. N)$

K-transformation: $\lambda x. y = K y ; (x \text{ and } y \text{ are different})$

I-transformation: $\lambda x. x = I$

M and N represents two different expressions.

We can use these reduction rules as an **example** to evaluate the above transformed expression by applying it to an argument say 4.

$$\begin{aligned}
 f\ 4 &= S(S((K\ +)(K\ 1)))\ I\ 4 \\
 &= S(K\ +)(K\ 1)\ 4\ (I\ 4) && \text{by S-transformation rule} \\
 &= K\ +\ 4\ (K\ 1\ 4)\ (I\ 4) && \text{by S-transformation rule} \\
 &= +\ 1\ (I\ 4) && \text{by K-transformation rule} \\
 &= +\ 1\ 4 && \text{by I-transformation rule} \\
 &= 5
 \end{aligned}$$

The benchmark examples outlined earlier can be handled in a similar way, but lead to very large combinator expressions. The relationship between combinator code size and the complexity of an expression can be viewed in terms of Antoni’s [18] observations.

Antoni [18] has found that the resulting code size in terms of number of combinators can be modelled by the following formula.

$$\begin{aligned} \text{code size} &= 3^m (n-1) + 1 \text{ if variables occur in term} \\ &= ((2n-1) 3^m + 1)/2 \text{ if variables do not occur in term} \end{aligned}$$

where n is the length of terms ie number of symbols on the right hand side of the source expression and m is the total number of variables. For example the expression

```
def f = lam x.lam y.lam z. x+y+z
```

have a length $n = 5$ (ie $x+y+z$) and the variable count $m = 3$ (ie x, y, z).

Observations on Alcal code are presented in the following table.

Expression	n	m	Code size
$\lambda x. x$	1	1	1
$\lambda xy. x+y$	3	2	19
$\lambda xyz. x+y+z$	5	3	109
$\lambda xyz u. x+y+z+u$	7	4	477
$\lambda xyz u v. x+y+z+u+v$	9	5	1928

Table : 4.1 Variable count with code size.

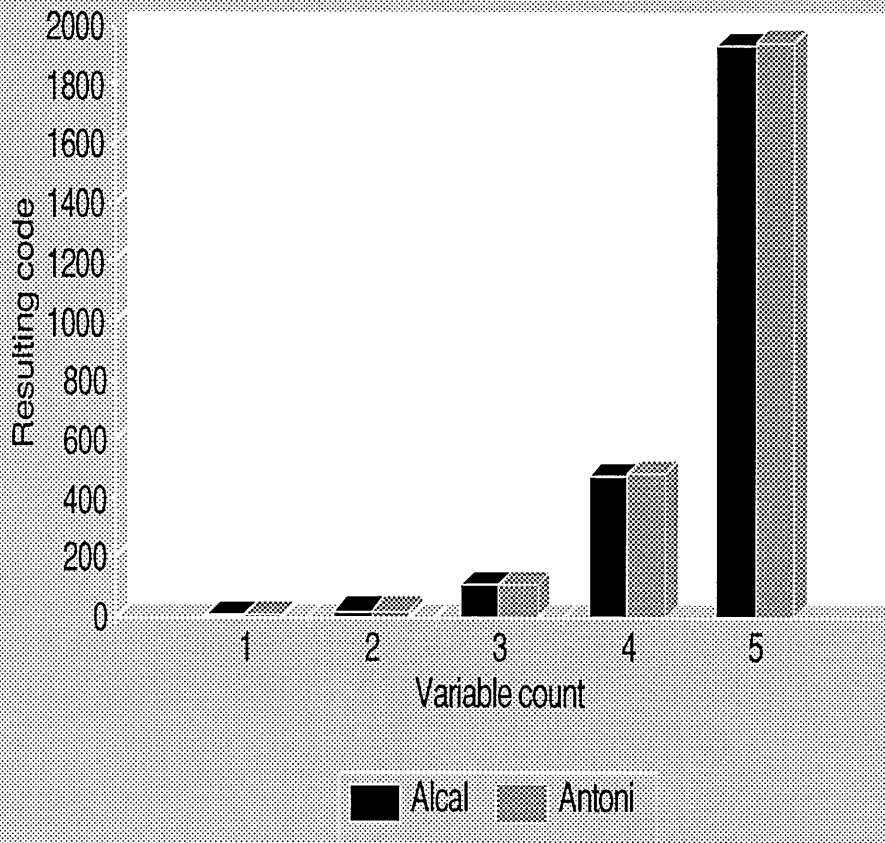
Table 4.1 shows the code produced by Alcal using various examples, a relationship between the variable count and the code size from the graphs given below may be given as

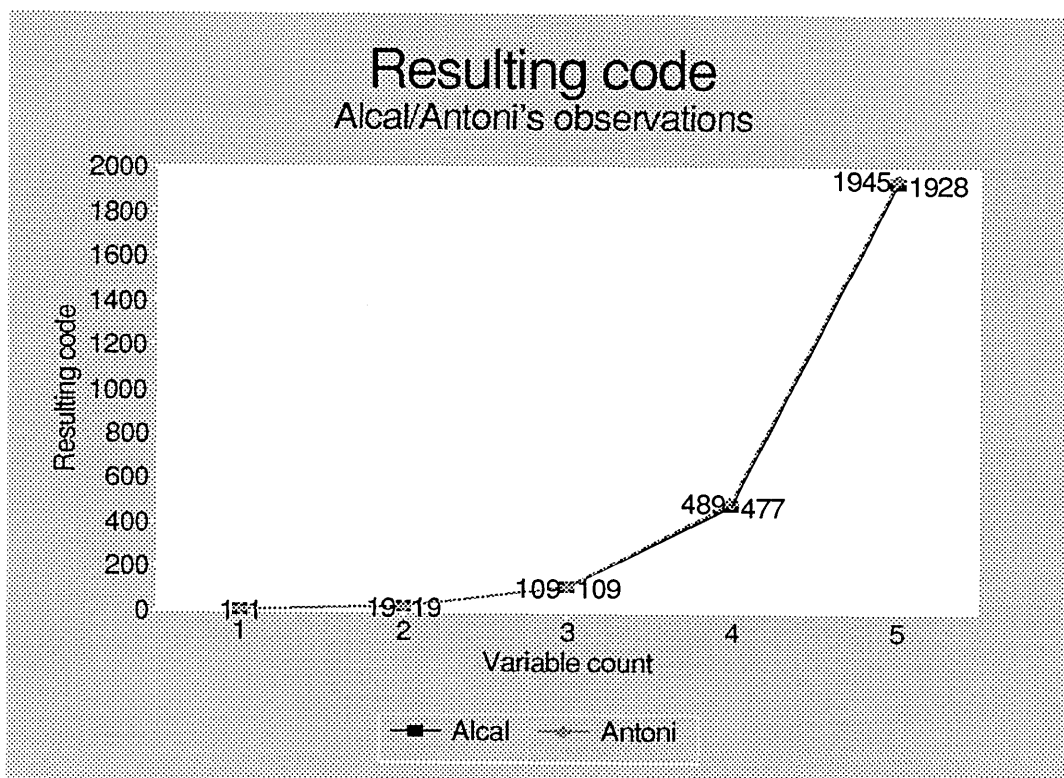
$$\text{code} = (\text{variable count})^{4.64}$$

The resulting code of Alcal and Antoni's [18] observation are given below: (Alcal's observation is around 2% better then Antoni's resulting code)

Resulting code

Alcal/Antoni's observations





The above table shows that the abstraction process produces long strings of combinators. Even very simple λ -expressions have long combinator expressions. It is possible to optimize this code on the grounds of efficiency.

4.3.1 Code optimization

It will be very much more efficient if the optimizations are performed during the translation process. To perform these optimizations, we have to introduce some extra combinators like B and C as described in chapter-3. The most important of these optimizations are motivated by Turner [65]. For example consider the following rule

$$S(K E1)(K E2) = K(E1 E2)$$

The left hand side of the above equation converts under β -reduction to the right hand side when the appropriate λ -expression is substituted for the combinator.

These optimization rules may be built in to the abstract function [22] as:

$$[x]x = I$$

$$[x]y = K y \text{ ;if } y \text{ is constant or variable not equal to } x.$$

$$\begin{aligned} [x] E_1 E_2 = & \text{if } [x]E_1 = K M \\ & \text{then if } [x]E_2 = I \\ & \quad \text{then } M \\ & \quad \text{else if } [x]E_2 = K N \\ & \quad \quad \text{then } K (M N) \\ & \quad \quad \text{else } B M [x]E_2 \\ & \text{else if } [x]E_2 = K N \\ & \quad \text{then } C [x]E_1 N \\ & \quad \text{else } S [x]E_1 [x]E_2 \end{aligned}$$

$$[x](E) = ([x]E)$$

In order to make the abstraction algorithm more efficient Turner introduced a number of further optimizations. These make use of extra combinators B' , C' and S' . These prime combinators have the following reduction properties:

$$B' f g x y = f g (x y)$$

$$C' f g x y = f (g y) x$$

$$S' f g x y = f (g y) (x y)$$

Turner describes these prime combinators in terms of combining the B-combinator with the following optimization rules:

$$S(B \ x \ y) \ z \ = \ S' \ x \ y \ z$$

$$B(x \ y) \ z \ = \ B' \ x \ y \ z$$

$$C(B \ x \ y) \ z \ = \ C' \ x \ y \ z$$

The motivation for introducing prime combinators significantly reduces the length of the resulting combinator code. This becomes clear on applying this algorithm to our benchmarks (described earlier).

The code produced for the benchmark function say Fibonacci function (fib) by introducing these optimizations in SKI and S', B', C' is:

$$\begin{aligned} \text{def fib} &= \lambda n. \text{if } n < 2 \text{ then } 1 \text{ else fib}(n-1) + \text{fib}(n-2) \\ &\Rightarrow ((S((S((S(K \text{ Cond}))((S((S(K <))I))(K \ 2))))(k \ 1))) \\ &\quad ((S((S(K +))((S(K \text{ fib}))((S((S(K -))I))(K \ 1)))))) \\ &\quad ((S(K \text{ fib}))((S((S(K -))I))(K \ 2)))) \end{aligned}$$

$$\begin{aligned} &\Rightarrow ((S(((C' \text{ Cond}))((C <)2))1))(((S' +) \\ &\quad ((B \text{ fib})((C -)1)))((B \text{ fib})((C -)2))) \end{aligned}$$

code reduced from 38 to 19.

Notice that this optimization saves execution time, because it requires fewer reductions to be done.

4.4 An active graph implementation

All combinator-based implementations, whether they use fixed combinators [65] or supercombinators [37, 41] can be characterised as *interpretive*, because the reduction and transformation of the combinator graph can be seen as an interpretation of the graph.

The ‘active graph’ implementation strategy in this thesis goes a long way towards answering Thompson’s [63] query-

‘Whether a true compiler for lazy functional programs can be produced ? ’

In Lingo, even the code sequences within a method are represented as objects. A language feature that allows the programmer to create code type objects, ‘module’, operates as follows:

We may write as:

```
y := module x [ ] { ^ x + 1 }
```

This specifies that variable y will be bound to a module containing the code

```
^ x + 1
```

The module expects to be supplied with a single argument (represented by functional parameter x) when its code sequence is executed.

Module objects respond to the message ‘**performWith:**’. This causes execution of the modules code sequence after substituting the message’s argument for the formal parameter. That is,

```
y performWith: 2
```

would give the result 3.

As can be seen, the module facility corresponds to a lambda-abstraction. For example

```
module x [] {^ x + 1}
```

corresponds to

```
lam x. x + 1
```

The performWith: message has as its counterpart in the lambda-calculus function application. For example

```
lam x. x + 1    5
```

can be represented in Lingo as:

```
(module x [ ] {^ x+1}) performWith: 5
```

which is similar to the Smalltalk's block context as:

```
[ :x | ^(x+1) ] value: 5
```

The primitive 'performWith:' is used for modules which return a specific result, it invokes its module with the argument (5 in this case) as its receiver. The result is the result of the module.

In Alcal the code for each combinator is available as part of the module representing it, each module provides an executable graph. Consider an example of combinator 'S'. The reduction rule of S is given as

$$S\ f\ g\ x = f\ x\ (g\ x) ;$$

which is represented in Lingo as:

```
{ ^ module f []
  { ^ module g of f []
    { ^ module x of g f []
  { ^(apply the S-reduction rule)}}}}
```

This means that the S combinator's module's code sequence is directly available to the processor. The modularity of these directly executable combinators offers an attempt to take advantage of the Rekursiv's hardware for method dispatching.

The use of such a mechanism ensures that

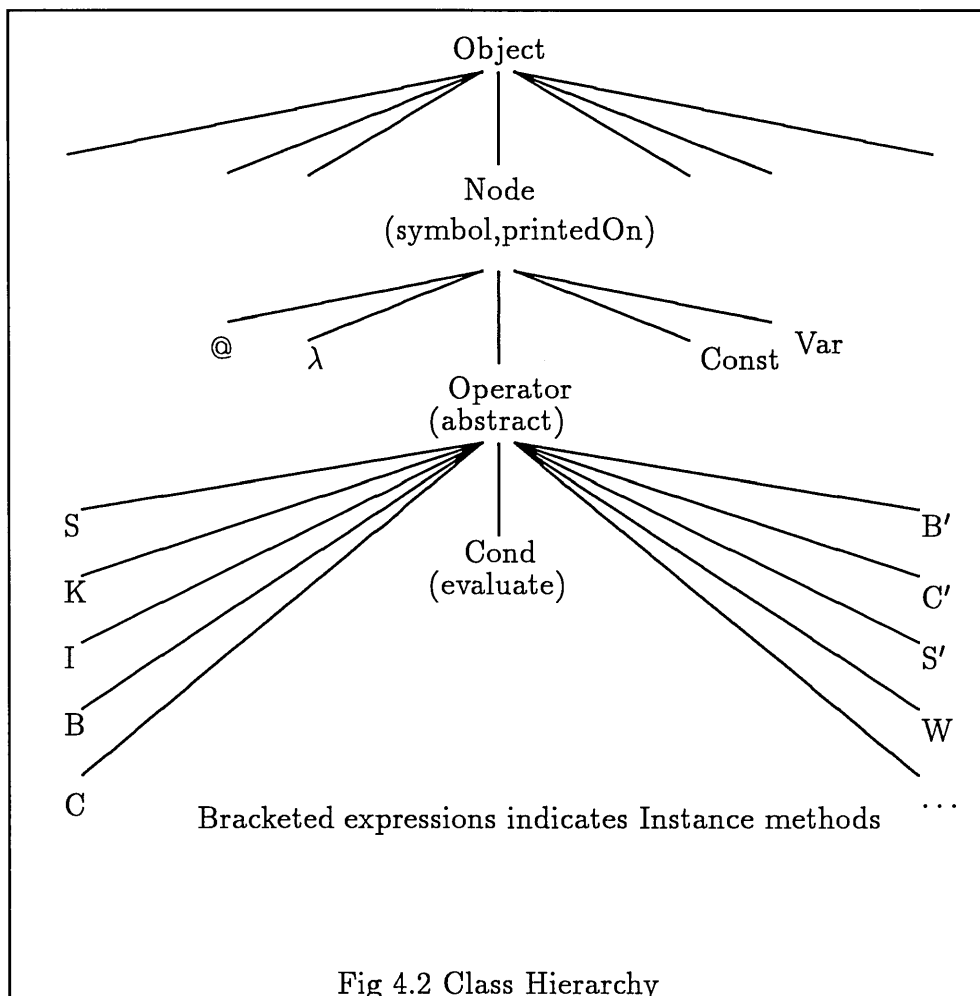
- Once a combinator has been specified at one level, its implementation at that design level and at lower levels can proceed independently, changes can be made at any time, so long it meets its specification.
- To add more combinators means simply to add few more objects at that level.

Because the module's code sequence in the Rekursiv is directly available to the processor, without resort to the object store, so caching the modules in this way may provide a considerable benefit in terms of performance.

This is the practical benefit of a good encapsulation mechanism, that certain changes can be made in the knowledge that the existing program will not break.

4.5 Class hierarchy and types of node

Alcal is a collection of several classes (nodes) as shown in class hierarchy figure 4.2. The combinators are stored as binary trees, with the nodes representing functional application, while the leaves of the tree are constants or S, K, I combinators or built-in operators like plus, minus etc.



This figure makes it clear that each combinator module has its own instance method called 'evaluate' while they all inherit the instance method 'abstract' from the module 'Operator'.

Each node of Alcal has its own class, and various type of node and leaves are defined as:

Application, Lambda, Variable, Constant, Operator

built in functions like *Plus, Minus, Divide, Times, Lessthan, Equal, Cond, S, K, I* are all defined as subclasses of *Operator* with superclass *Node*. The class hierarchy of *Plus* is:

$Plus \subset Operator \subset Node \subset Object$

For example, the expression $\lambda x. (1+x)$ is a combinator of four type of nodes ie *Operator, Variable, Constant* and *Application*. The expression $((+ 1) x)$ can be represented as

```
Application of:(Application of:Plus new to:
(Variable named:'x')) to:(Constant value:1)
```

When we want a *Plus* we need only send the message '*Plus new*', here '*new*' is defined as a class of *Plus* which creates a new *Plus* for an interactive session.

To implement Lists, a few more nodes like *Cons, Hd, Tl, Nil, And, Or, Not* have to be defined. The code of these combinators is given in Appendix-B.

4.6 Evaluation strategy

Consider an example of combinator '+', the combinator '+' is represented by a class *Plus* with methods:

```

class method      : new
instance method  : abstract
                  : evaluate
                  : printedOn.

```

We are concern here with the method ‘evaluate’ only. In this method, the primitive ‘performWith:’ is used in a way to adopt the lazy evaluation.

(left evaluate) performWith : right;

The performWith primitive has the effect of an unary message sending operation. It was mentioned earlier in this chapter that, the primitive ‘performWith:’ is used for modules which return specific results, it invokes its module with the argument as its receiver. Considering the method ‘evaluate’ in its simpler form:

```

{ ^ module x []
{ ^(x + 1)}}

```

The result is the result of the module. The argument (there must be at least one argument, for example a ‘+’ needs to be applied to two numbers before it is reducible) is then examined to determine its class and the method dictionaries of this and the classes in its inheritance chain are searched to identify the meaning of the message selector. Once the required method has been found, it is examined to determine whether it corresponds to a primitive or a method. In case of primitive the appropriate microcode is executed otherwise the method’s module code is executed. Similarly in S combinator reduction having the reduction rule as:

$$S\ f\ g\ x = fx\ (gx) ;$$

the primitive ‘performWith:’ will only be invoked when all the arguments or receiver to S are there ie $f\ g\ x$. Otherwise it can not be invoked by sending it any other message because it has no receiver.

Chapter 5

Performance Evaluation

This chapter is concerned with

- the measurement and comparison of the performance of various implementation strategies (SK combinators and λ -lifting) by timing the execution of individual combinators/supercombinators and the execution of some benchmark functions.
- the determination of the contribution to performance resulting from the underlying Rekursiv architecture by comparing the Rekursiv implementations against implementations using Smalltalk-80 on a (conventional) RISC processor (the IBM RS6000)
- the determination of the contribution to performance made by using classes to represent objects as opposed to instances of classes.

5.1 Description of experiments

To describe the performance evaluation experiments, the execution of the benchmark function *Ex0* is taken as an example running on Rekursiv. Details of other

benchmarks are listed in Appendix-C.

Ex0 is a simple function with one argument (successor function):

```
lam x. x + 1;
```

Ex0 was executed repeatedly for a period of 100 seconds. The amount of time spent on executing *Ex0* includes:

- Code generation
- code evaluation
- Overheads

which are discussed below.

The test program for timing the execution of the benchmark function *Ex0* for a period of 100 seconds is given as:

```
Tex0 is Object
{
  []
run := 0;
Time setTimeOfDay:0; /* initialise time counter to 0 */
while (Time getTimeOfDay < 100)
  do {
    (Ex0 test:4);

    /* Ex0 is a subclass of Operator. */

    run := run + 1;
  }.
}
```

5.1.1 Code generation

The interpretation of *Ex0* results in the following combinator code:

$$(((C\ Plus)\ 1)\ 4)$$

Which is represented in Lingo as:

Ex0 is Operator

```
{
[]
test:x []
{
^((Application of:(Application of:(Application of:
(C new) to:(Plus new)) to:(Constant value:1)) to:
(Constant value: x)) evaluate);
}.
}
```

Instances of combinator objects *C*, *Plus*, *Constant* (two in this case), *Application* (3 in this case) are created. That is, seven objects are generated. The reduction rule for combinator '*C*' is:

C is Operator

```
{
[]
evaluate []
{^ module f []
  {^ (module g of f []
    {^ (module x of g f []
      {^(Application of:(Application of:f to:x) to:g)
        evaluate;))}})}.
```

This generates two more instances of *Applications*, which takes the total number of objects created per run up to 9. That is, five *Application* objects, two *Constant* objects, one *C* object and one *Plus* object. This means that time spent on executing *Ex0* includes nine object creation times.

Object creation time can be easily determined (see below Table:1.2).

5.1.2 Code evaluation

Time spent on the process of combinator code evaluation includes:

1. time spent on executing Lingo code which includes:
 Application objects (5 times)
 Constant objects (2 times)
 C object
 Plus object.
2. time spent on executing the Lingo code which includes the executions of the primitive arithmetic operator '+’.

Time spent on executing the Lingo code for evaluating a combinator object (say ‘C’) can be determined from the number of evaluations which take place during a 100 second run, as shown in the following Table:1.1;

<i>Code</i>	<i>time per run</i>
<i>C evaluate</i>	13.8 μ -sec

Table 1.1: Evaluation time for various combinator objects like S, B, C, Sprime etc on the Rekursiv architecture.

Similarly the object creation time can also be determined as (Table:1.2);

<i>Code</i>	<i>time per object creation</i>
<i>C new</i>	31.53 μ -sec

Table 1.2: Object creation time for various combinator objects like S, B, C, Sprime etc on the Rekursiv architecture.

The creation of other combinator objects can be determined in a similar way. Object creation time (except for the *Application* object) is similar for all of them. The object creation time for an *Application* object, (for example '*Application of:1 to:2*') is measured as 38.6 μ -sec.

The evaluation of an *Application* object uses the Lingo primitive 'performWith'.

```
{^ ((left evaluate) performWith: (right));}
```

Here the operation 'left evaluate' will always returns a combinator object, so the evaluation time of an *Application* object's 'left evaluate' remains more or less constant (13.8 μ -sec). Timing an *Application* object can be performed as:

```
{(module x [] {^x}) performWith: 2)
```

which altogether takes 17.91 μ -seconds, out of which the left part ie

```
{(module x [] {^ x})}
```

takes 0.71 μ -seconds, which gives the time spent on the primitive 'performWith' as 17.2 μ -sec.

So, the evaluation time for an *Application* object may be given as:

$$13.8 + 17.2 = 31 \mu sec$$

Object creation time and object reduction (evaluation) time for all the combinator objects are listed in Appendix-D1.2.

5.2 Overheads

Overheads on the Rekursiv arise primarily from two sources:

1. garbage collection
 2. Object Squeeze (for detail see chapter:1 ‘Objects on the Rekursiv’).
- Garbage collection

Time spent on garbage collecting on the Rekursiv can be determined by invoking the Lingo instruction ‘Object gc’. Data on running the Test program (see above) for a period of 2 seconds with and without invoking ‘gc’ is given below in Table:1.3;

<i>No. of runs per 2 sec with <u>one</u> gc</i>	<i>No. of runs per 2 sec with <u>no</u> gc</i>	<i>time spent on gc</i>
1533	2327	684 milli-sec

Table 1.3: Garbage collection (gc) time on the Rekursiv architecture.

This means that Rekursiv takes an extra 684 milliseconds per gc during the benchmark executions. Other experiments confirms this value.

- Object squeeze

The squeeze time varies with implementations and with the representation of combinator objects as Class representation or Instance representations of the specific combinator object.

Since squeeze time is related with the object creation time, the time spent per squeeze for each benchmark with different implementation techniques will be different, it may be determined as:

$$\text{Time spent}/(\text{gc} + \text{squeeze}) = \frac{(\text{measured time per run} - \text{calculated time per run})}{\text{Total Number of (gc+squeeze)}}$$

The total number of (gc + squeeze) can either be observed or calculated.

We shall see that the measured and calculated times for runs without 'gc + squeeze' are identical (to within $\pm 2\%$). This provides a justification for assuming that the discrepancy between measured and calculated runs with 'gc + squeeze' is caused solely by the overheads involved in squeeze and gc.

In table D1.2 (Appendix-D), a complete list of various combinator objects timing are given. It includes object creation/reduction timings of combinator objects, the execution time of the primitive arithmetic operators like +, - and the garbage collection time on the Rekursiv and RISC architectures.

5.3 Experiments

We have to take account of 3 different parameters, resulting in 8 (2^3) different experiments for each of four benchmarks.

1. Implementation technique:

- SKI combinators ('standard' combinators)
- λ -lifting (ie supercombinators)

2. Architectures:

- Rekursiv (using Lingo)
- RISC (using Smalltalk-80)

3. Object-oriented style:

- Combinator objects represented as classes
- Combinator objects represented as instances of classes

The following questions are addressed:

1. Is λ -lifting better than SKI-combinator implementation, if the answer is yes then by how much?
2. Is Class representation of combinator objects better than Instance representation, if yes then by how much?
3. How does the Rekursiv differ from a RISC machine?

The results which will be used to address the above questions are given below. This table of results gives the execution performance of some benchmark functions on RISC and Rekursiv architectures. The number of executions for a time period of 100

seconds have been recorded for Lambda-lifting and SKI-combinator implementations using Class and instance representations:

Abbreviations used are:

Ex0, Ex1, Fact, Ack : Benchmarks (for details see Appendix-C1.1)

RE, RI : denotes Rekursiv and RISC architectures

LI : implementation of Lambda-lifting with Instance representation

LC : implementation of Lambda-lifting with Class representation

SI : implementation of SKI-combinators with Instance representation

SC : implementation of SKI-combinators with Class representation

<i>Exec./100 sec</i>	<i>Ex0</i>		<i>Ex1</i>		<i>Fact 3</i>		<i>Ack 1 1</i>	
<i>Architecture</i>	<i>RE</i>	<i>RI</i>	<i>RE</i>	<i>RI</i>	<i>RE</i>	<i>RI</i>	<i>RE</i>	<i>RI</i>
<i>LI</i>	203371	31640	70128	10594	9876	1755	5999	962
<i>LC</i>	220659	35154	84827	12213	13294	2013	7821	1144
<i>SI</i>	107560	16059	32900	5214	5232	940	2776	611
<i>SC</i>	124040	17747	39949	5761	6239	983	3349	649

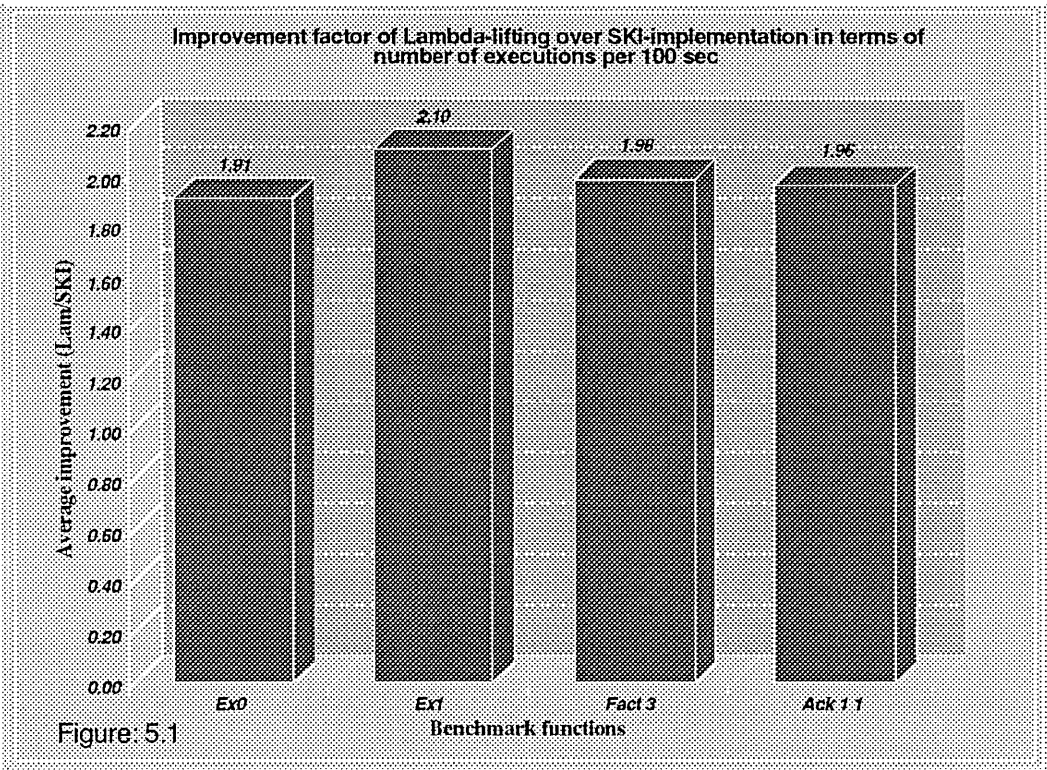
Table:1.4 Number of executions of some benchmarks functions per 100 seconds, for λ -lifting and SKI-combinator implementations using Class representation and Instance representation of combinator objects on the Rekursiv and RISC architectures.

5.3.1 λ -lifting Vs SKI-implementation

The improvement of λ -lifting against SKI-implementation on the RISC and the Rekursiv architecture is given below in a table and as a bar-graph.

<i>Benchmark functions</i>	<i>Ex0</i>	<i>Ex1</i>	<i>Fact 3</i>	<i>Ack 1 1</i>
<i>Improvement factor of λ-lifting over SKI-combinators, (Average over both benchmarks, both Class/Instance representations).</i>	1.91	2.10	1.98	1.96

Table:1.5 Improvement factor of λ -lifting over SKI in terms of number of executions per 100 sec



The above histogram represents the improvement of λ -lifting against SKI-combinator implementations averaged over RISC and Rekursiv architectures representing the combinator objects as Class and Instance representations.

Is Lambda-lifting is better then SKI-implementation ?

We observed:

<i>mean improvement of lambda over SKI (over all benchmarks/architectures /Class representations)</i>	<i>range (ie least to most improvement)</i>
1.99	1.91 to 2.10

This can be explained as follows:

We have already seen that, in the process of λ -lifting, any function can be translated into supercombinator form by adding extra formal parameters corresponding to the free variables appearing in the function body. Briefly the performance advantage of supercombinators over SKI-combinators is largely due to the differences in ‘grain sizes’ of the execution steps.

In fact a single supercombinator performs the work of several SKI-like combinators. Consider the example for benchmark function ‘Ex0’. It is simply a successor function:

$(\lambda x. x + 1) 4$

Comparing the number of nodes (which means number of reductions) produced for Ex0, (ignoring the argument ‘4’) gives:

<i>Code with standard combinators</i>	<i>Code with supercombinators</i>
$((C +) 1)$	$(+ 1)$

The λ -lifting process generates only one supercombinator which is made up of two combinators ('Plus' and a 'Constant' combinator), while the SKI-combinator process has to perform three combinator reductions ('C', 'Plus' and the 'Constant' combinator).

The total number of reductions on each benchmark for both implementations can be given as:

<i>Benchmark function</i>	<u><i>SKI-combinator</i></u>		<u><i>Lambda-lifting</i></u>	
	<i>Total No. of reductions</i>	<i>No. of application reductions</i>	<i>Total No. of reductions</i>	<i>No. of application reductions</i>
<i>Ex0</i>	9	5	5	2
<i>Ex1</i>	26	17	14	7
<i>Fact 3</i>	212	137	93	48
<i>Ack 1 1</i>	257	178	89	49

Table:1.6 No. of Application node reductions per benchmark evaluation.

This means that in some cases one supercombinator may perform the work of several standard combinators. The cause of this reduction may be explained by considering the number of reductions performed during the execution of each benchmark function: *Ex0*, *Ex1*, *Fact 3*, *Ack 1 1*. The breakdown of reductions for benchmark *Ex1* is detailed in Table: 1.7. Tables of rest of the benchmarks are given in Appendix-D1.3.

<i>combinator objects</i>	<i>SKI-combinator implementation</i>	<i>Lambda-lifting implementation</i>
<i>Plus</i>	1	1
<i>Minus</i>	1	1
<i>Constant</i>	3	3
<i>B</i>	1	0
<i>C</i>	2	0
<i>Cprime</i>	1	0
<i>Application</i>	17	7
<i>Ex1sc1</i>	0	1
<i>Ex1sc2</i>	0	1
<i>total reductions</i>	26	14

Table: 1.7 Number of combinator reductions for *Ex1*. *Ex1sc1* & *Ex1sc2* denotes supercombinators. *S*, *B*, *C*, *Sprime*, *Cprime* denotes standard combinators.

For the improvement of λ -lifting over SKI-implementation, a regression analysis of the results of λ -lifting on SKI-implementation is given below. Since the computations involved in regression analysis are quite extensive, we have used MINITAB [57].

Here we are monitoring the contribution to performance resulting from the λ -lifting implementation against SKI-combinator implementation, while other parameters like the selection of object-oriented styles or the selection of the architectures may vary.

Regression analysis is helpful in determining the relationship between variables. Given the amount of time required for one variable (SKI-value), we can use the regression equation (see below) to predict the amount of time required for other

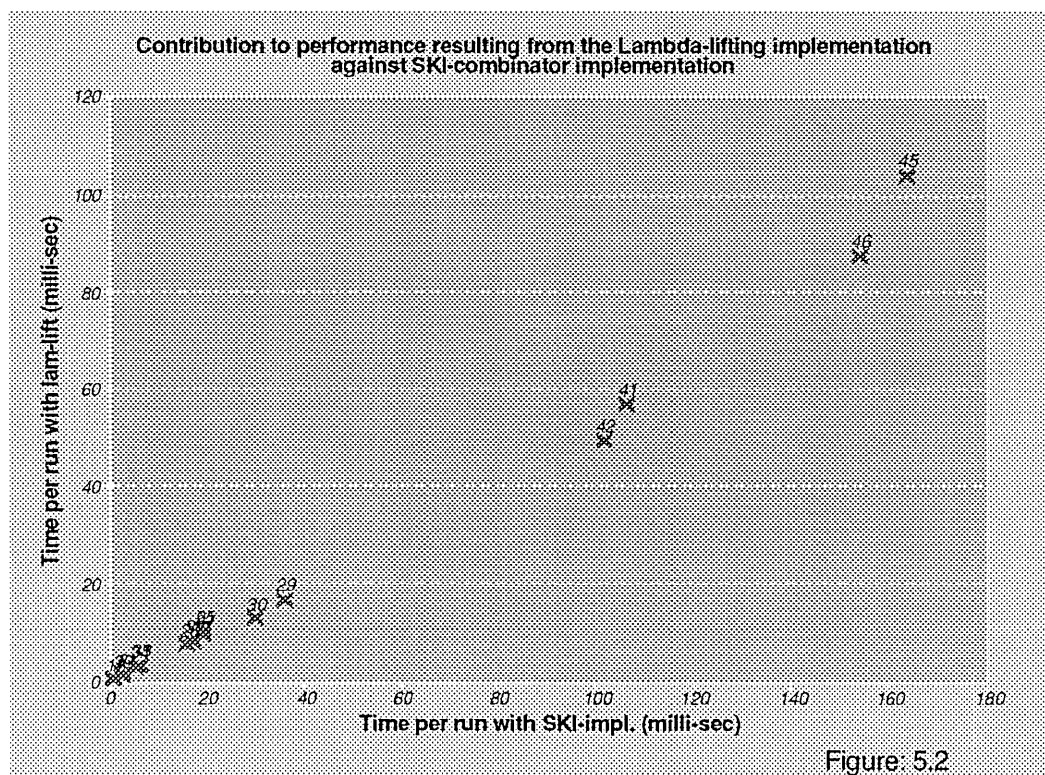
variables (λ -value) from a linear function of the SKI-value. That is, regression analysis find values of b_0 (*y-intercept*) and b_1 (*slope*) in the straight line equation

$$\text{predicted value of } \lambda \text{ time} = b_0 + b_1 * (\text{SKI time})$$

The SKI-times are shown on the *x-axis* and λ -times on the *y-axis*. The table of 16 pairs of observations is given below (see also appendix D1.7 where the details of each these observations are given with their interactions in a larger table of 48 observations).

Techniques	Observation numbers (details are listed on appendix D1.7)															
SKI	17	18	33	34	21	22	37	38	25	26	41	42	29	30	45	46
λ	19	20	35	36	23	24	39	40	27	28	43	44	31	32	47	48

The data are graphed in Figure: 5.2.



The fitted regression line using the method of least squares is given by:

$$\text{predicted value of } \lambda \text{ time} = -1771 + 0.587 * (\text{SKI time}) \mu\text{sec}$$

This regression method accounts for 98.73% of the variance ($R^2=0.9873$). Where 'R' is the correlation between λ -time (observed) and λ -time predicted and R^2 is the percent variance explained by the regression. R^2 is a statistic that measures the validity of the model, it ranges up to 1, with 1 being optimal. Such a large value of R^2 suggests that there is a close relationship which is approximately linear between SKI and λ . For each unit increase in SKI, there is an average increase in λ of 0.587, that is the slope of line is 0.587

However there is some suggestion of a systematic departure from linearity: the points with largest values of SKI and λ lie above the fitted line. It appears that λ -time increases with SKI-time a little faster than linearly, and this effect can be seen at the high end of the range of values in this data set. It may be concluded that the improvement of λ -lifting is decreasing as we go along to more complex computations.

Clearly the relationship expressed by the regression line can not be assumed to extend beyond the range of observed values; at the high end non-linearity may become important and at the low end we see that the regression line would predict a negative value of λ when SKI is zero.

This shows that the implementation using λ -lifting is faster than SKI-combinator implementation. The major reasons for the improvement contributed by λ -lifting is are:

- smaller number of reductions
- fewer application nodes
- less intermediate code generated

λ -lifting requires 12 fewer reductions than standard combinators for *Ex1*, including 10 *Application* node reductions; which in turn reduces the amount of garbage collection. The evaluation of one *Application* node reduction on the Rekursiv takes around 31 μ -seconds (for timing measurements see section 5.1.1). The ten additional applications required by the SKI-implementation contribute 310 μ -sec to its execution time.

The execution of a benchmark may be broken down as follows:

```

Execution = Combinator reductions
           + Object creations
           + Overheads

```

The proportion of time spent on overhead activities has been calculated (see timings in section 5.2). This is least for *Ex0*, SKI-implementation (40%) and greatest for *Ack* with λ -lifting (61%). These overheads apply further suppression to the differences between the two implementations.

As well as reductions, object creation plays a major part in the performance of each benchmark. Object creation time is of the same order of magnitude as reduction time (for timings see section below). The number of objects created for the SKI-implementation of *Ack 1 1* is 187 and for the λ -lifting implementation is 165. The improvement of λ -lifting over SKI for *Ack 1 1* is therefore suppressed relative to that of *Ex1* where the number of objects (26 and 14) are identical with the numbers of reductions in both cases. This can be seen in the figure:5.2, where the *Ack 1 1* points (45 and 46) are relatively higher than a straight line fitted through the other points.

The above figures for object creations apply to instance representations. The effect is similar for class representations (see following section), although the number of objects created are somewhat less.

The λ -lifting implementation produces less intermediate code than the SKI-combinator implementation so it uses less transient storage and this is another advantage of supercombinator reduction. More intermediate code means more allocation of cells in the memory. This increases the load on the number of storage accesses required contributing to the observation that SKI-implementation is generally slower than lambda-lifting implementation.

In summary, the improvements of λ -lifting over SKI-combinator performance are due primarily to a smaller number of combinator reductions and less intermediate code being produced. The contribution of object creation is less clear cut, as is that of overheads, which are of the same order of magnitude in both cases.

5.3.2 Class representation Vs Instance representation

In the process of evaluation, the creation of an object is performed by calling on a class to create a new object. For example, 'S new'. This creates an instance of S.

In the case of combinator objects like *S*, *K*, *I*, *Plus* etc, the instances of *S*, *K*, *I* and *Plus* are created during the translation process. Since there is no private data associated with these objects, it is possible to use classes to represent them. For example we can simply use '*S*' rather than '*S* new'. This avoids storing multiple instances of the same class (thus saving a number of object creations).

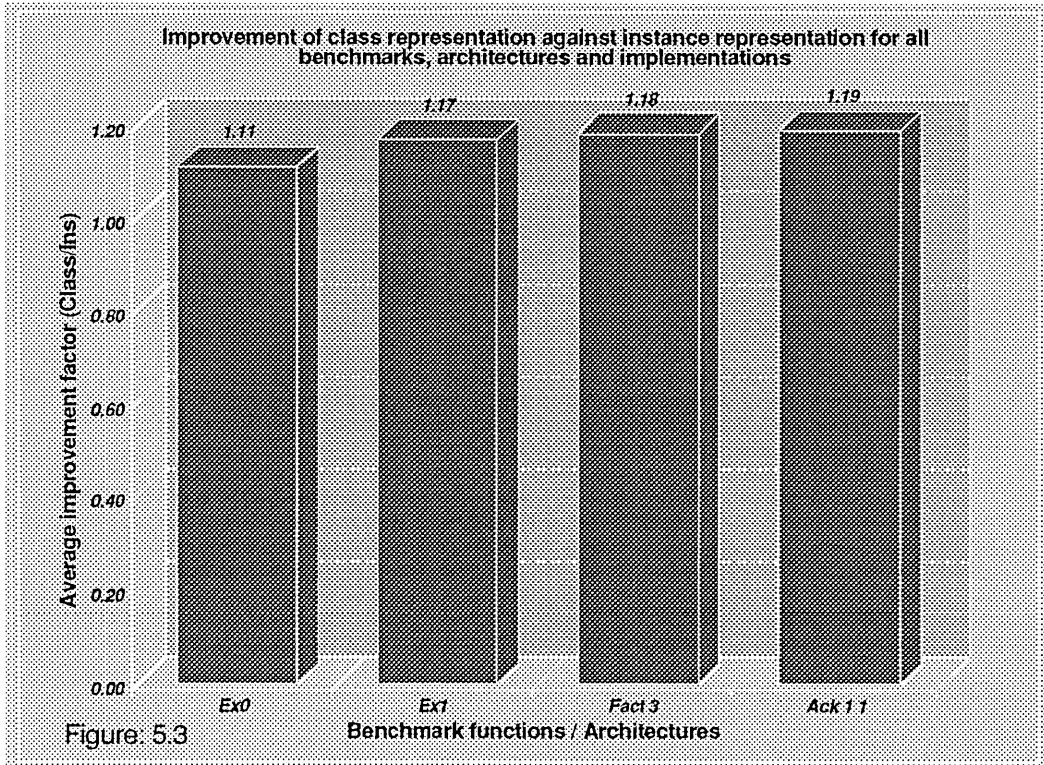
The evaluation of a *Constant* object, on the other hand returns the value with which it is associated. *Application* objects are also associated with data, so it is not possible for *Application* objects and *Constant* objects to be represented as classes.

The evaluation of *Application/Constant* objects creates a similar number of objects (one in this case) on both implementations (standard and supercombinators). The benefit of saving a number of object creations is to be found during the evaluation of both standard and supercombinator implementations.

The improvement of class representation against instance representation of combinators is given below in a table and as a bar-graph for each of the benchmarks.

<i>Benchmark functions</i>	<i>Ex0</i>	<i>Ex1</i>	<i>Fact 3</i>	<i>Ack 1 1</i>
<i>Improvement factor of Class representation over Instance representation, (Average over both benchmarks, and on both implementations).</i>	1.11	1.17	1.18	1.19

Table:1.8 Improvement factor of Class and Instance representation of objects averaged over RISC and Rekursiv architecture with λ -lifting and SKI-combinator implementations in terms of number of executions per 100 sec.



The above bar-graph represents the improvement of Class representation against Instance representation of combinator objects averaged over RISC and Rekursiv

architecture with λ -lifting and SKI-combinator implementations.

From the table above we observe:

<i>overall improvement factor of Class over Instance representation</i>	<i>range over benchmarks (ie least to most improvement)</i>
1.16	1.11 to 1.19

Using classes to represent combinators, rather than instances of classes, gives a small but significant improvement in number of executions per 100 seconds. The number of reductions involved in evaluating a function is the same for both instance and class representation but the number of objects created varies. The following table gives the saving of the number of objects created by selecting class representation of combinator objects for all the benchmark functions.

<i>No. of objects created</i>	<u><i>SKI-implementation</i></u>		<u><i>Lambda-lifting</i></u>	
<i>Benchmark function</i>	<i>instance representation</i>	<i>class representation</i>	<i>instance representation</i>	<i>class representation</i>
<i>Ex0</i>	9	7	5	4
<i>Ex1</i>	26	20	14	10
<i>Fact 3</i>	100	89	80	54
<i>Ack 1 1</i>	187	159	165	130

Table : 1.9 The number of objects created.

The cause of this improvement can be explained in terms of the number of reductions and number of objects created for 100 second execution of a benchmark

function (say *Ex1*) using both representations (Class/Instance) with SKI-combinator implementation on the Rekursiv.

combinator objects	<u>Instance representation</u>		<u>Class representation</u>	
	reductions	objects created	reductions	objects created
<i>Plus</i>	32900	32900	39949	0
<i>Minus</i>	32900	32900	39949	0
<i>Constant</i>	98700	98700	119847	119847
<i>B</i>	32900	32900	39949	0
<i>C</i>	65800	65800	79898	0
<i>Cprime</i>	32900	32900	39949	0
<i>Application</i>	559300	559300	679133	679133
<i>total</i>	855400	855400	1038674	798980

Table: 1.10 No. of objects created and No. of reductions for Ex1 (100 sec run) with SKI-implementation on the Rekursiv. Note that the number of reductions per 100 sec run is improved since less time is wasted on the creation of unnecessary objects.

Tables for other benchmarks involving number of object creations with corresponding number of reductions are given in Appendix-D1.4.

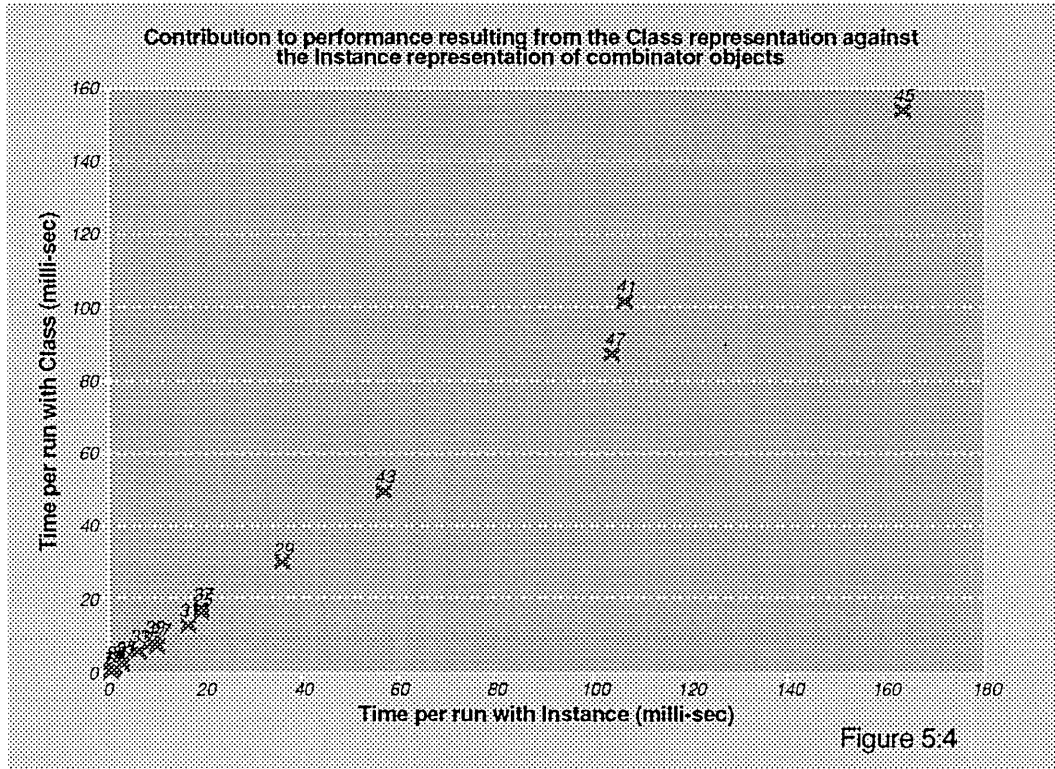
This saving of object creation causes both the RISC and Rekursiv architectures to spend less time in allocating cells in memory and so gives an overall improvement factor of 1.16 for a 100 second run.

To represent the improvement of Class representation over Instance representation, a regression analysis of the results of class representation on instance representation is given below. The table of observations (16 set of observations) is listed below. The details of each observation with their interactions are given in detail (48 observations) in appendix D1.7.

Here we are monitoring the contribution to performance resulting from Class representation against Instance representation of combinator objects. The other factors like implementation techniques or architectures may vary.

Styles	Observation numbers (details are listed on appendix D1.7)															
Class	18	34	20	36	22	38	24	40	26	42	28	44	30	46	32	48
Instance	17	33	19	35	21	37	23	39	25	41	27	43	29	45	31	47

It was mentioned earlier that by regression analysis the relationship between the variables (Instance and Class) can be established. We can use regression equation to predict the value of Class representation from a linear function of the Instance value. The data are graphed in figure:5.4.



The fitted regression line using the method of least squares is given by:

$$\text{predicted value of Class rep. time} = -1127 + 0.9269 * (\text{Instance rep. time}) \mu\text{sec}$$

The regression model in this case accounts for 99.64% the variance ($R^2 = 0.9964$). This large value of R^2 again suggests that there is a close relationship which is approximately linear between Class and Instance representation and that Instance-values is a good linear predictor of the Class-value. For each unit increase in Instance value, there is an average increase in Class value of 0.9269 (slope).

It is clear from figure:5.4 that most data points lie close to the regression line. The high end and low end both show linearity so this relationship expressed by the regression line may be extended beyond the range of observed values and we may predict a Class value from a linear function of the Instance value with some degree of confidence.

Using classes to represent combinator objects as opposed to instances is a step forward in reducing the overall overhead on both architectures (RISC and Rekursiv).

5.3.3 How does Rekursiv differ from RISC?

The performances of different implementations like lambda lifting and SKI-combinator with various object-oriented styles (Class and Instance representations) have been discussed in the above two sections. This section is involved with the architectural issue by comparing the relative performance of the Rekursiv and RISC architectures in terms of the implementation of functional languages. This includes some statistics on the benchmark results to analyse some of the claims made by the designer of the Rekursiv architecture.

The following issues are addressed:

- The contribution (improvement) of various implementation techniques (λ -lifting and SKI-combinator) on the Rekursiv and on the RISC architectures.
- The contribution of various object-oriented styles (Class and Instance represen-

tation of combinator objects) on the Rekursiv and on the RISC architectures.

- Comparison the performance of functional language implementations on the Rekursiv and on the RISC processors is difficult because of differences between execution platforms and operating system environments. However we may generate a performance base line by executing a sample program which gives similar results on both processors.
- Some statistics on the results of benchmark functions which also reveal to what degree built-in features of the implementation language (for example garbage collection) can be exploited.

5.3.3.1 Improvement with the Rekursiv/RISC architectures over different implementations

The overall improvement of λ -lifting and SKI-implementation on the Rekursiv and the RISC architectures are given below.

<i>Architectures</i>	<i>Overall improvement factor of λ-lifting and SKI-implementation over all benchmarks/object-oriented style</i>	<i>Maximum improvement of λ-lifting and SKI-implementation</i>
<i>Rekursiv</i>	2.05	2.34
<i>RISC</i>	1.92	2.16

Table 1.11: Improvement factor of λ -lifting and SKI-combinator implemetation using the Rekursiv and RISC architectures in terms of number of executions per 100 sec over all benchmarks function and different object-oriented styles.

5.3.3.2 Improvement with the Rekursiv/RISC architectures over different (object-oriented) styles

The improvements of class representation against instance representation of combinator objects over the Rekursiv and RISC architectures are given as:

<i>Architectures</i>	<i>Overall improvement of Class and Instance representation over all benchmarks/techniques</i>	<i>Maximum improvement of Class and Instance representation</i>
<i>Rekursiv</i>	1.214	1.35
<i>RISC</i>	1.115	1.21

Table 1.12: Improvement factor of Class representation and Instance representation of combinator objects over all benchmarks function using λ -lifting and SKI-combinator implementation on the Rekursiv and RISC architectures.

5.3.3.3 Generating a performance base line for the Rekursiv and the RISC

A benchmark program has been coded in Smalltalk/Lingo and C and executed on the Rekursiv and the RISC.

This benchmark evaluates an arithmetic expression

$$(11 + (10 + (9 + (8 + (7 + (6 + (5 + (4 + (3 + (2 + 1)))))))))))))$$

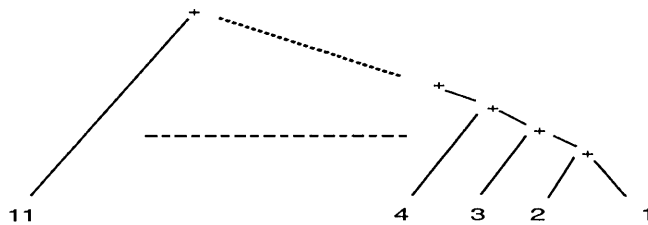


Figure: 5.5

which can be represented as a tree (using instances of classes to represent nodes in the Smalltalk/Lingo code, and using union structs (variant records) in the C code.

The timings are in milliseconds for a single evaluation of the benchmark.

<i>Architectures</i>	<i>Smalltalk/Lingo</i> <i>milli-sec</i>	<i>C</i> <i>milli-sec</i>
RISC	1.7	0.034
Rekursiv	0.3	0.044

Table 1.13: Performance of Smalltalk/Lingo and C on the RS6000 and the Rekursiv

This shows that the timings of the C program on the Rekursiv and the RISC are of the same order of magnitude. By taking the performance of C as a baseline, the relative performance with Lingo on the Rekursiv is faster (more then 7 times) than Smalltalk on the RISC. That is, the performance behaviour of C is similar on both architectures and there is an improvement when executing a Smalltalk type language on the Rekursiv. This confirms that the Rekursiv does, indeed, reduce the performance bandwidth (see section 1.6)

For the performance behaviour of executing functional languages on the RISC and on the Rekursiv architectures, we have performed two analyses:

- regression analysis
- analysis of variance

Time taken per benchmark execution on the RISC and on the Rekursiv are recorded in μ -sec.

5.3.4 Regression analysis

The regression analysis is performed to determine the contribution of functional language implementations to performance on the Rekursiv and the RISC architectures.

Four benchmarks with two implementation techniques on two object-oriented styles gives 16 observations on each architecture. Corresponding pairs of observations are listed below, the details of each observation with their interactions are given in appendix D1.7.

We are monitoring the contribution to performance resulting from the Rekursiv architecture against RISC architecture, while the other factors like the style of implementation or technique of implementation may vary.

Rekursiv Observations 17 to 32

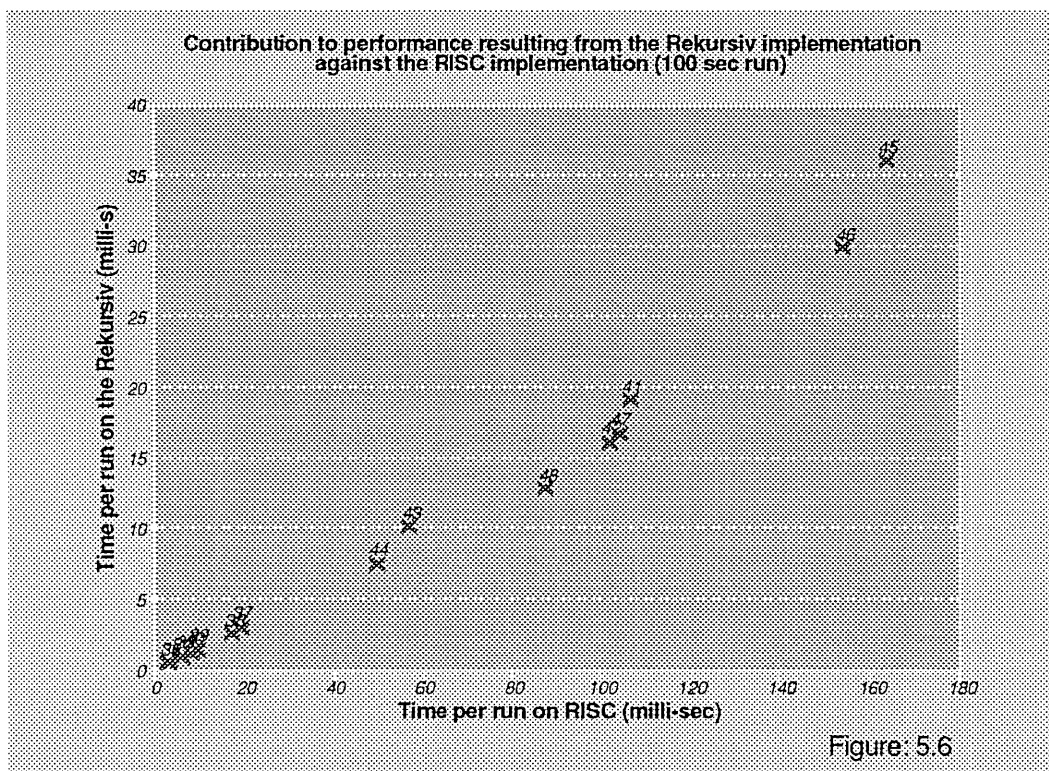
RISC Observations 33 to 48

The regression analysis is used here to predict the Rekursiv-value from a linear function of the RISC-value.

Unusual observations:

Observation numbers		
45	46	48

The observations (above three) give some indication of not fitting very well on the line.



The fitted regression line is given as:

$$\text{predicted value for the Rekursiv} = -1002 + 0.1953 * (\text{RISC value}) \mu\text{sec}$$

This regression model accounts for 96.91% the variance ($R^2 = 0.9691$). This large value of R^2 similar to previous two regression analysis suggests that there is a close relationship which is approximately linear between the Rekursiv and RISC, and that RISC values is a good linear predictor of the Rekursiv value. For each unit

increase in RISC, there is an average increase in Rekursiv value of 0.1953 (the slope of the line = 0.1953).

The figure:5.6 shows that most data points (except a few high values) are scattered about the straight line. The points with largest value of RISC and Rekursiv lie above the fitted line, so at high values, the unusual observations 45, 46, 48 suggest a departure from linearity. The Rekursiv increases with RISC a little faster than linearly, this effect can be seen at the high end of the range of values in this data set. At the low end the regression line would predict a negative value of Rekursiv when RISC value is zero, clearly this relationship between Rekursiv and RISC can not be assumed to extend beyond the range of observed values.

This means that on the Rekursiv the plot is falling a little faster than linearly which degrades the performance of Rekursiv on high values (caused by the Rekursiv's 'gc+squeeze' involvement, see below). This means that the advantage of the Rekursiv is decreasing as we go to more complex computations. The Rekursiv overall reduces the performance bandwidth, resulting in a considerable (subject to a limited amount of computations, see below) contribution to performance resulting from the underlying Rekursiv architecture.

These performance experiments have been carried out for a 100 sec run, by executing a benchmark function for a large number of times. The very large number of executions means a large number of object creations, and hence an increased load on garbage collection.

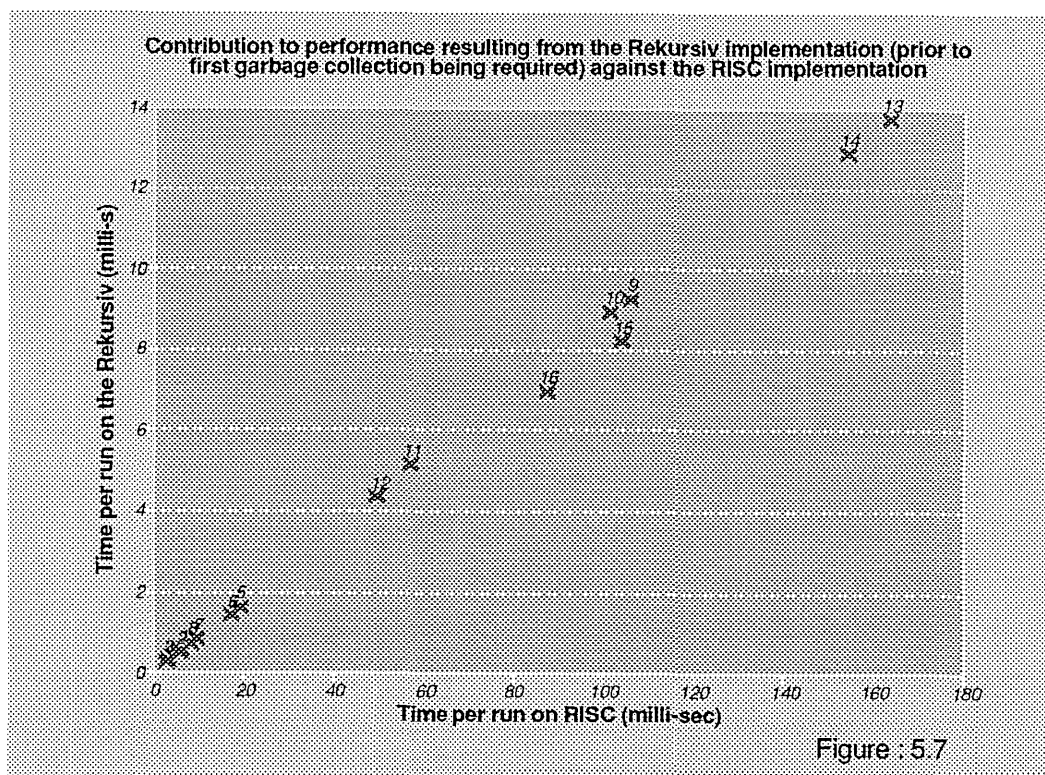
5.3.5 The effect of garbage collection

On the RISC there is a constant garbage collection overhead irrespective of the benchmarks execution (see appendix-D1.5). This overhead is always there whether we execute these benchmarks for a small or for a large amount of time.

<i>Architectures</i>	<i>mean overhead factor over all benchmarks/implementation techniques/object-oriented styles</i>	<i>range over benchmarks (ie least to most overhead factor)</i>
<i>RISC</i>	0.08 (8 %)	0.075 to 0.085
<i>Rekursiv when no garbage collection is required</i>	0.45 (45%)	0.39 to 0.62
<i>Rekursiv prior to first garbage collection</i>	0.0127 (1.2%)	0.00 to 0.02

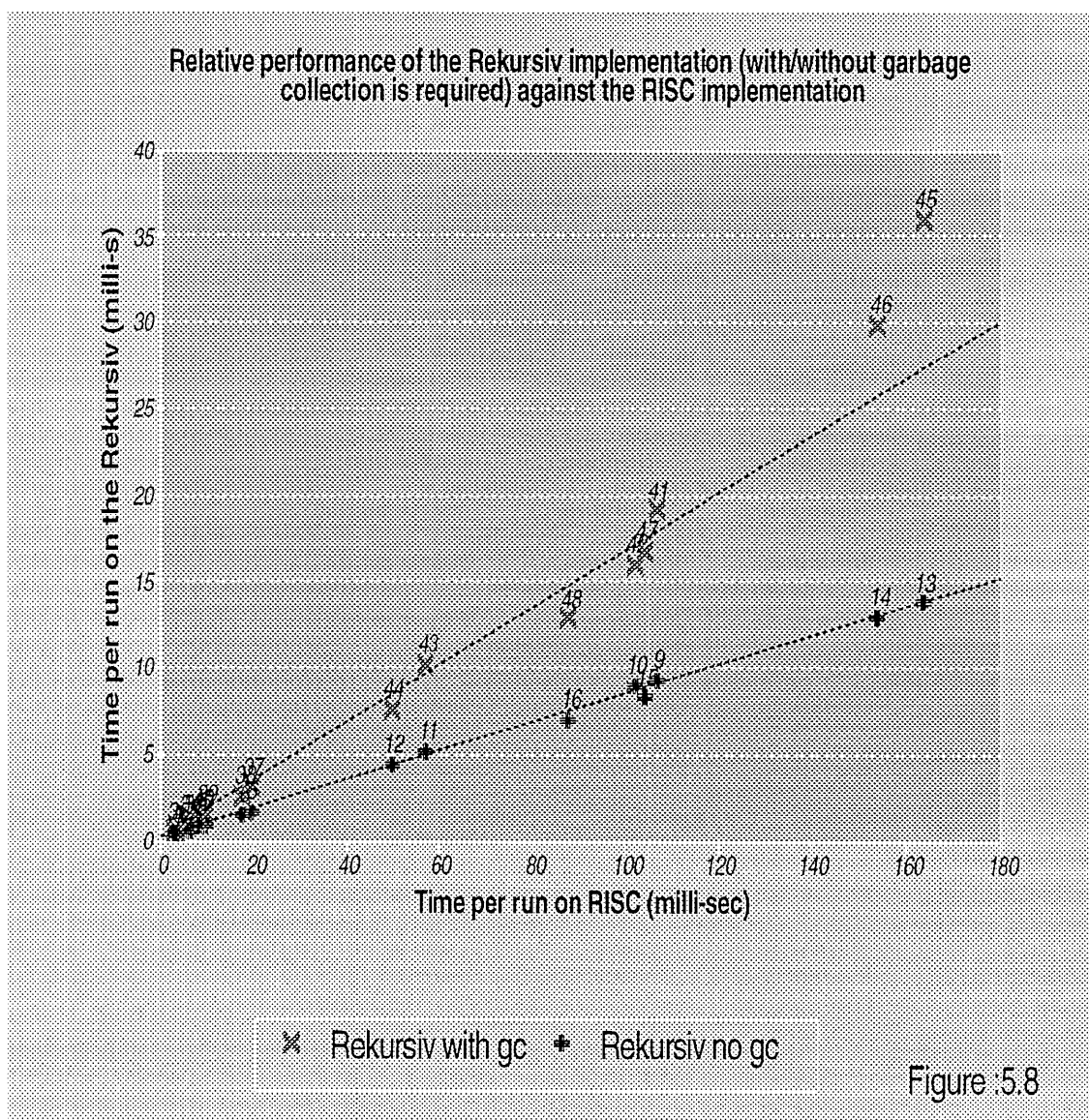
Table 1.14: The garbage collection overhead factor of the RISC and the Rekursiv architectures averaged over all benchmarks with λ -lifting and SKI-implementation in terms of with/without requiring garbage collection.

The resource allocation strategy of Rekursiv is discussed in chapter-1. In the case of the Rekursiv experiments show that before garbage collection is required, the overhead remains minimal. (the overhead factor of executing various benchmarks on the Rekursiv in detail (see appendix-D1.6)). These observations were made prior to the first garbage collection. If we compare the results for the Rekursiv when no garbage collection is required, with the result for the RISC, we observe that there is a linear relationship. With no garbage collection on the Rekursiv, figure:5.7 shows that there is an even greater linear performance improvement. For details of each observation shown in figure:5.7 see appendixD1.7.



On the Rekursiv a large number of objects may be created (subject to a maximum limit), most of them once created are retained in the persistent object store. The number of objects potentially accessible may thus be very large. Relatively few objects will be required at any one time, and these may be heavily used. Observation shows that the most complex benchmark we have used (*Ack 1 1*) creates only 187 objects per run, and these objects are used heavily during the execution. Since the paging system of the Rekursiv can hold upto 65535 objects at any one time the overall design of Rekursiv assumes relatively low paging rates. Only very large programs are likely to involve garbage collection and squeezing during their execution.

The following figure:5.8 represents the relative performance of Rekursiv on RISC with/without garbage collection on the Rekursiv.



The fitted regression line equations of with/without garbage collection is given as: (Note: on the graph 'gc' indicates garbage collection)

$$\text{predicted Rekursiv value with gc} = -1002 + 0.1953 * (RISC \text{ value}) \mu sec \quad (5.1)$$

$$\text{predicted Rekursiv value prior to gc} = 49.81 + 0.0837 * (RISC \text{ value}) \mu sec \quad (5.2)$$

The regression model (5.1) accounts for 96.91% the variance ($R^2 = 0.9691$).
The regression model (5.2) account for 99.74% the variance ($R^2 = 0.9974$).

The large value of R^2 in both cases suggests that there is a close relationship which is approximately linear between RISC and the Rekursiv. This is especially true for the model 5.2 where R^2 is close to the optimum value of 1. This confirms that model 5.2 is more linear (even on higher values) than the first model (5.1).

The relationship expressed by model 5.2 indicates that this regression line can be assumed to extend beyond the range of observed values with some degree of confidence: the high end and the low end are both linear and so its regression equation can be used to predict the amount of time required for Rekursiv-value from a linear function of the RISC-value.

The figure:5.8 shows that the performance improvement on the Rekursiv is considerable when no garbage collection is required but the improvement degrades somewhat with excessive garbage collection. In fact this garbage collection is performed with the squeeze fault and time of squeeze varies with the executions of different benchmarks (see chapter-1 and the first section of this chapter).

5.3.6 Analysis of variance

This section is concerned with the analysis of variance for multi-way (architectures, techniques, styles, benchmarks) balanced designs. The model (anova) deals with three order interactions. It calculates all expected means, estimates variance components and plots the residual and fitted values for both main effects and their interactions.

5.3.6.1 Factors in model

Our model is designed with four factors.

1. Four benchmark functions

- 1 = *Ex0*
- 2 = *Ex1*
- 3 = *Fact 3*
- 4 = *Ack 1 1*

2. Three machines

- 1 = Rekursiv disregarding garbage collection and object squeeze
- 2 = Rekursiv with garbage collection and object squeeze
- 3 = RISC

3. Two implementation techniques

- 1 = SKI-combinator
- 2 = λ -lifting

4. Two object-oriented styles

- 1 = Instance representation
- 2 = Class representation

A total of 48 observations ($2*2*3*4=48$) has been recorded for this model. All timing measurements are in μ -sec. The table of 48 observations with their interaction factors (see above) and a complete history of various analysis paths are given in appendix.D1.8.

All the interactions are indicated with asterisks(*). For example, 'instclas*skilam' is the interaction between factors instclas and skilam. It is mentioned earlier that this model deals with third order interactions so for main effects, 2-factor interactions and 3-factor interactions see appendixD1.8.

Finally after removing the third order insignificant effect the analysis of variance for log time is given in the Anova table:1.15.

The terms used for columns in anova table:1.15 are described as:

Source : description of interactions of various factors

DF: degrees of freedom, it is one less than the number of effects e.g. there are 4 benchmark effects so DF is 3.

$$Total\ SS = \text{sum of } (observation - \text{mean of all observations})^2$$

e.g. Total SS = (First obs. - mean of 48 obs.)² +(second obs.....

SS : sum of squares is partitioned by the analysis of variance.

MS: mean square (SS/DF)

F : Fisher's F statistic,

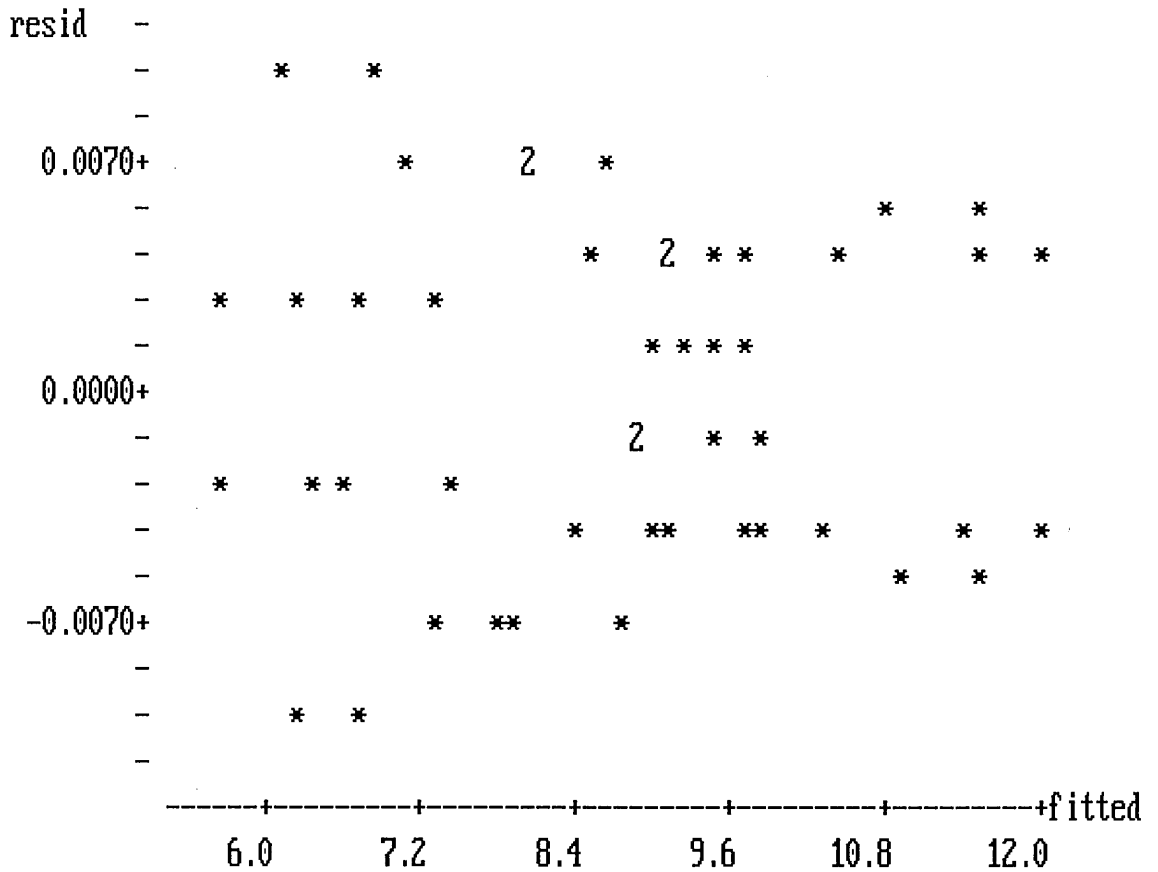
$$Fisher's\ F\ for\ an\ effect = \frac{\text{the } MS\ for\ the\ effect}{\text{the } MS\ for\ error}$$

Error : error is the estimate of variance not explained by any of the effects in the table.

P : probability that *F* would be as large as this if the effect were zero. If $P < 0.05$ then we say the effect is significant at 5%.

<p style="text-align: center;">Anova Table: 1.15</p> <p style="text-align: center;">Analysis of variance for time (3rd order insignificant effects removed)</p> <p style="text-align: center;">(see appendix-D1.8 for details of the analysis)</p>					
<i>Source</i>	<i>DF</i>	<i>SS</i>	<i>MS</i>	<i>F</i>	<i>P</i>
<i>instclas</i>	1	0.2353	0.2353	1421.85	0.000
<i>skilam</i>	1	5.2917	5.2917	3.2E+04	0.000
<i>benchmk</i>	3	89.7335	29.9112	1.8E+05	0.000
<i>machine</i>	2	51.7687	25.8843	1.6E+05	0.000
<i>instclas*skilam</i>	1	0.0089	0.0089	53.47	0.000
<i>instclas*benchmk</i>	3	0.0036	0.0012	7.15	0.012
<i>instclas*machine</i>	2	0.0165	0.0083	49.88	0.000
<i>skilam*benchmk</i>	3	0.0168	0.0056	33.87	0.000
<i>skilam*machine</i>	2	0.0174	0.0087	52.59	0.000
<i>benchmk*machine</i>	6	0.0917	0.0153	92.34	0.000
<i>instclas*skilam*benchmk</i>	3	0.0090	0.0030	18.19	0.001
<i>instclas*benchmk*machine</i>	6	0.0072	0.0012	7.27	0.007
<i>skilam*benchmk*machine</i>	6	0.0404	0.0067	40.68	0.000
<i>Error</i>	8	0.0013	0.0002		
<i>Total</i>	47	147.2421			

The residual plot for the table:1.15 is also given: (for rest of tables and residual plots see appendixD1.8).



Residual Plot (Fig: 5.9)

The residual plot (figure: 5.9) shows a tendency for variance to decrease from left to right. The four biggest residuals belong to the observations at benchmark 1 on machine 2. These four observations are not very well explained by the model.

We removed the *instclass*machine*skilam* interaction because it was insignificant. However, it seems possible that even though this interaction is not significant there is some complex relationship between these factors which our new model fails

to deal with.

The column 'P' on the above final table indicates a probability. It is mentioned earlier in table:1.15 that if $P < 0.05$ then we say the effect is significant at the 5% level. As can be seen, all main effects are significant: so observed differences among means for the three machines, for example, are unlikely to be due to random variation. The same applies to differences among means observed for the levels of the other factors. All but one of the 2-factor interactions are also significant: so for instance the difference between machines 1 & 2 (or any other pair) is not the same at all benchmarks.

Two of the 3-factor interactions are also significant. These results together suggest that the differences among the factor levels are important for all factors and also that the relationships among the factors are quite complex.

Chapter 6

Conclusions

The aim of this thesis was to investigate the ability of an object-oriented architecture to provide a general platform for the implementation of functional languages. This thesis has presented a number of new results in the field of functional programming, including the relative performance of two different implementation techniques for functional languages implemented in a variety of object-oriented styles on two different hardware environments.

The following objectives have been met (see section 1.6):

1. Implementation of functional languages using Lingo on the Rekursiv.

The entire implementation is straightforward and clear. We have not dealt with environments and closures etc, because we strictly follow the normal order in a lazy manner.

The ease with which functional programs are developed in an object-oriented paradigm via Alcal was due to the expressivity of the object-oriented language used (Lingo), and its polymorphism.

One of the striking features of the Rekursiv architecture is its internal parallel executions of various processors, many processors are built with a common

memory bus, each having a different task to perform in parallel.

The dominant factor limiting the performance of any system (architecture) is the overhead of moving data. It is possible to map any abstract evaluation model onto a conventional computing system by programming. New architectures especially the object-oriented architecture (Rekursiv) can be justified if they result in improved performance. In addition, the Rekursiv significantly simplifies programming, gives good run time diagnostics, error messages and a user controllable garbage collection routine. Implementing Alcal on the Rekursiv allows the derivation of efficient code because various optimization techniques can be applied in a systematic way at each level of transformation.

The primary motivation for using Rekursiv architecture is that we believed it can significantly improve performance in evaluating functional languages and second motivation is the ease of programming and code generation.

2. Performance analysis on λ -lifting and SKI-combinator implementation.

The practical functional programming system such as that using an 'active graph' can be improved by basing it on supercombinator implementation rather than SK-set of combinators. Supercombinators result in a much larger grain size for reduction-steps and, compared to SKI-combinator implementation, requires fewer reductions for evaluations, therefore supercombinators reduce the number of nodes and the number of combinators executed and so speed up the system.

One factor which contributes to the improvement of λ -lifting implementation over SKI-combinator implementation is the resource allocation of the Rekursiv. More object creations (more SK-like combinators means more objects) puts more demand on resource allocation and this degrades the overall performance.

3. Identification of major issues in terms of general implementation strategies.

The earliest implementation of Alcal (Alcal0, say) used string matching to differentiate between node types. There was a single class Node, to represent all the elements in a graph. On instantiation, an instance variable was loaded with a string denoting its nature (S, K, Application, Plus,...etc). The evaluation of a graph required the inspection of this instance variable to determine the appropriate reduction action. This is analogous to using a variant record type in a conventional language.

Later implementations used the class system to perform this differentiation implicitly. In the case of Lingo this allowed the hardware type checking to come into play. This resulted in a major (~ 10 fold) performance improvement.

4. Comparison of the Rekursiv implementation against RISC implementation.

We have simulated the performance sensitivity with respect to variation of individual parameters for several benchmarks. As a check on the accuracy of our simulation, we compare the results with measured performance on the Rekursiv and RISC architectures.

Comparing two environments for reducing similar kinds of combinator object is difficult because of different execution platforms and operating system environments. We found it difficult to predict the speed of graph reduction on RISC as compare to the Rekursiv. We compare these machines with respect to conventional C program. We found that the variation in performance on both machines with conventional C program is similar. But there are unexpected variations in performance when functional programs have been implemented on these architectures; specifically with regard to the memory management. Nonetheless, we were able to determine the relative performance of RISC and the Rekursiv. Based on these comparisons, Rekursiv appears to be a faster

combinator graph reducer (a factor of more than 7 times faster).

Our selection of benchmarks is rather limited, due to the unavailability of a standard benchmark suite. However, whatever the size of the benchmark (small or large), they generally consume only a small amount of memory space for object creation, and once an object is created, it is heavily used.

From the results of this study we can conclude that combinator-graph reduction using Lingo on the Rekursiv gives reasonable performance improvement (about 7 times) compared with Smalltalk on the RISC.

5. Contribution of the style of object-oriented implementation to performance.

The average improvement factor of class representation against instance representation of combinator objects is about 1.16. It is small but significant improvement in number of executions per 100 sec. It reduces the number of objects creations per run.

It can be concluded that using classes to represent combinator objects may reduce the overall overhead on both architectures and hence gives better performance in terms of execution time.

6. Effect of garbage collection.

The experiments performed with 100 sec runs shows some unexpected parameter interaction in the performance (eg see the upper values in figure: 5.8). We therefore explore variations which avoid excessive consumption of computational resources. We use the same benchmarks, but with a somewhat smaller number of runs so as to keep the memory traces down to a manageable size (the executions are performed for less than three seconds). In this way a complete program execution is performed in all cases on both architectures, and garbage collection is not invoked at all for any of these runs on the Rekursiv even with *Ack 1 1*.

The results from these experiments confirm that, in the case of the Rekursiv the unexpected parameter interaction is due to the requirement of garbage collection performed by the system. Without garbage collection on the Rekursiv these interactions are reduced to an average of about 1%. In the case of RISC there was no change in the unexpected parameter interactions, it remains constant (around 8% of the computation time) throughout the entire computation.

The above simulation results are an important architecture design tool. However, in order to establish some confidence in the simulation results, analysis of variance was made between the results of executions on the RISC and the Rekursiv architectures.

An analysis of variance leads to the conclusion that Rekursiv reduces the performance bandwidth and hence gives a considerable improvement for the implementation of function languages over other architectures.

The users and developers of reducers for standard or supercombinator graph reduction (with full optimizations) can use the results of our experimentation to aid in selecting a hardware platform that will perform well. Furthermore the approach used is reasonably fast and simple to implement on other object-oriented languages (eg Smalltalk) and so provides a relatively easy way to obtain an implementation of a lazy functional language.

Other strategies

The G-machine provides a good basis for a realistic implementation of a functional language. Combinators can be compiled directly into executable code to give a better performance. In the G-machine and T-machines, garbage collection is required during executions. Furthermore, the G-machine, requires yet another function 'doAdmin' which checks the memory size after each step. In contrast the garbage collection is performed by the underlying hardware on the Rekursiv. This

automatic garbage collecting facility relieves the programmer of the responsibility of keeping track of the above mentioned references. There are no results available from the above machines to compare with; in fact it is not yet a completely mature practical approach, but is the subject of a lot of research and may well have a promising future.

6.1 Future work

A great deal of further research is required to develop a full understanding of the Rekursiv architectural issue in terms of microcoding in practical functional programming.

In particular, the bulk of overheads involved in resource allocation of cells in the Rekursiv can be eliminated by employing the use of compact objects. Since compact objects do not occupy pager table slots or memory locations, Harland has claimed that performance can be dramatically improved if compact objects are used. It remains to be seen whether combinator objects are accessible to this transformation. The manipulation of compact objects involves the use of the Rekursiv at a deeper level than Lingo affords, and is beyond the scope of this thesis.

Our benchmark programs are fairly standard with respect to the literature in functional programming language implementation, but it must be realized that it is an extremely limited set. Even though all our benchmark programs allocate a small amount of memory, we have seen some different behaviour. To see some clear relative behaviour we need many more experiments. A widely acceptable suite of standard benchmarks is needed.

In our discussion we have avoided many complex issues like polymorphism and pattern matching because this thesis was intended as a vehicle for relative performance of functional languages on various architectures. However for more complex

functional languages these may allow some elegant solutions. These issues can be investigated in future research.

6.2 Contribution of this thesis

The main results presented in this thesis are:

- Performance evaluation of λ -lifting Vs SKI-combinator implementation.
- Performance evaluation of different object-oriented styles.
- Performance evaluation of novel hardware (Rekursiv) compared with traditional (Risc).
- An active graph model for the reduction of λ -expressions.

Bibliography

- [1] America, Pierre (1989). *Issues in the Design of a parallel O. O. Language*, Formal Aspects of Computing (1989) 1: 366-411, 1989
- [2] Argo, Guy (1989). *Improving the three instruction machine*, Computing Sc. Department, The University, Glasgow.
- [3] Augustsson, L. (1984). *bf A compiler for lazy ML*, Proceedings of the 1984 ACM symposium on Lisp and Functional programming, Austin, Texas, August 1984, pp.218-227.
- [4] Backus, John (1978). *Can Programming be liberated from the von Neumann Style?*
A Functional Style and its algebra of Programs, Communication of ACM, V-21, No. 8, August 1978.
- [5] Belof, B., McIntyre, Duncan and Drummond, Brian (1988). *OBJEKT data book*, Document Number LSD0036, Linn Smart Computing Limited, Glasgow, 1988.
- [6] Bird, R. S. (1987). *A formal development of an efficient supercombinator compiler*, Science Computer Program (Netherlands), Vol 8(2), 1987, pp 113-137.
- [7] Birtwistle, G. M., Dahl, O. and Nygaard, K. (1979). **Simula BEGIN**, Chartwellbratt.

- [8] Blair, G., Gallagher John, Hutchison David and Shepherd D (1991). **Object-oriented languages, systems and applications**, Pitman Publishing (1991).
- [9] Budd, Timothy (1991). **Object-oriented programming**, Addison-Wesley, 1991.
- [10] Burge, W. H. (1975). **Recursive Programming Techniques**, Addison-Wesley, 1975.
- [11] Burstall, R. M., MacQueen, D. B. and Sanella, D. T. (1980). *Hope: an experimental applicative language*, CSR-62-80, May 1980, Department of Computer Science, University of Edinburgh.
- [12] Cheese, Andy (1987). *Combinatory code and a packet based computational model*, SIGPLAN Notices, Vol 22, No 4, 1987, pp 49-58.
- [13] Curry, H. B. and Feys, R. (1958). **Combinatory Logic**, Vol 1, North Holland Publishing Company, Amsterdam, 1958.
- [14] Church, A. (1941). *The Calculi of Lambda Conversion*, Ann. of Math. Studies, Vol 6, 1941.
- [15] Darlington, J., Field, A. J. and Pull, H. (1985). *The unification of Functional and Logic Languages*, Report NB 35-0012, Department of Computing, Imperial College of Science and Technology, London SW7 2BZ, February 1985.
- [16] Davie, A. J. T. and Morrison, R. (1981). **Recursive Descent Compiling**, Ellis Horwood, 1981.
- [17] Dennis, J. B. (1979). *The varieties of data flow computers*, Proc. 1st Int. Conf. Distributed Computing system, France, Oct 79, pp430-439.
- [18] Antoni, Diller (1988). **Compiling functional languages**, 1988.
- [19] Drummond, Brian (1988). *Hades data book*, Document Number LSD0039, Linn Smart Computing Limited, Glasgow, 1988.

- [20] Elsley, Ian (1989). *Kontroller user Manual (Rekursiv), Document NO. LSD0054*, Linn Smart Computing Limited, Glasgow, 1989.
- [21] Fairbairn, Jon (1985). *Design and Implementation of a Simple Typed Language Based on The Lambda Calculus*, Technical Report No. 75, May 1985, University of Cambridge, Computer Laboratory.
- [22] Field, Anthony J. and Harrison, Peter G. (1988). **Functional Programming**, Addison-Wesley, 1988.
- [23] Finn, Simon (1987). *Hoisting: Lazy Evaluation in a cold climate*, LNCS NO. 250, TAPSOFT '87.
- [24] Kennaway, J. R. and Sleep, M. R. (1983). *Novel architectures for declarative languages*, Software and Microsystems, Vol. 2, No. 3, June 1983.
- [25] Frederick, S. (1990). *O. O. Programming Applied to Prototype Workstation*, Software-Practice and Experience, V-20(9), pp 887-898, Sept 1990.
- [26] Goldberg, A. (1983). **Smalltalk80: the language and its implementation**, Addison-Wesley 1983.
- [27] Glaser Hugh, Hankin Chris and Till David (1984). **Principles of Functional Programming**, Prentice/Hall International, 1984.
- [28] Glaser, H. and Hayes, S. (1986). *Another implementation technique for applicative languages*, ESOP 86: European Symposium on Programming proceedings, pp 70-81, Springer-Verlag, 1986.
- [29] Gordon, M. J. and Milner, A. J. and Wadsworth, C. P. (1979). Edinburgh LCF.
- [30] Hailpern, Brent (1989). *Comparing two functional programming systems*, IEEE Trans on Software Engineering vol-15, No. 5, May 1989.
- [31] Harland D. M. (1988). **Rekursiv object-oriented computer architecture**, Ellis Harwood Limited, 1988.

- [32] Harland, D. M. (1989). *A guide to Lingo* Linn Smart Computing Limited, Document No. LSD0057, Edition-2, August 89.
- [33] Henson, Martin C. (1987). **Elements of Functional Languages**, Blackwell Scientific Publications, London, 1987.
- [34] Hindley, J. Roger (1985). *Combinators & Lambda calculus: A short outline*, Lecture notes in Computer Science, No: 242, Combinators and Functional Programming Language, pp 105-122, May 1985.
- [35] Hudak (1984). *A Combinator Based Compiler for Functional Language*, Proc 11th ACM symposium on Principles of Programming Languages, pp 121-132, 1984.
- [36] Hughes, John (1982). *Graph Reduction with Supercombinators*, Technical Monograph PRG-28, June 1982, Oxford University Computing Laboratory Programming Research Group.
- [37] Hughes, R. J. M. (1982). *Super combinators: a new implementation method for applicative languages*, Proc ACM, Symposium on LISP & functional programming, pp 1-10, 1982.
- [38] Hughes, R. J. M. (1985). *Lazy memo function*, Proc Conference on Functional Programming Languages and Computer Architecture, LNCS, Vol 201, pp 129-146, 1985.
- [39] Johnsson, Thomas (1984). *Lambda Efficient compilation of lazy evaluation*, Proceedings of the SIGPLAN'84 symposium on compiler construction, Montreal, Canada, June 1984, pp. 58-69.
- [40] Johnsson, Thomas (1985). *Lambda Lifting: Transforming Programs to Recursive Equations*, LNCS, Conference on Functional Programming Language and Computer Architecture, Vol 201, pp 198-203, 1985.

- [41] Johnsson, Thomas (1987). *Compiling Lazy Functional Languages*, A Dissertation for the Ph. D. in Computer Sciences at Chalmers University of Technology, 1987.
- [42] Jones, Simon L. Peyton (1985). *An introduction to fully lazy supercombinator*, Lecture notes in Computer Science, No:242, Combinators & functional programming language, pp 176-208, May 1985.
- [43] Jones, S. L. Peyton (1987). **The Implementation of Functional Programming Languages**, Prentice/Hall, 1987.
- [44] Jones, Simon L Peyton and Lester, David (1990). *A modular fully lazy lambda lifter in Haskell*, CS Report series CSC-90/R17, CSC June 1990, Department of Computer Science, University of Glasgow, Glasgow G12 8QQ.
- [45] Jones, S. L. Peyton and Lester, David (1992). **Implementing Functional Languages**, Prentice/Hall, 1992.
- [46] Jones, T. C. (1991) **Applied software measurement**, McGraw Hill, 1991.
- [47] Kaehler, Ted and Patterson, Dave (1986). **A Taste of Smalltalk**, First Edition, W W Norton & Company, 1986.
- [48] Kelly, Paul (1989). **Functional Programming for Loosely-coupled Multiprocessors**, Pitman, 1989.
- [49] Koopman, Philip J., Jr (1989). *A fresh look at combinator graph reduction*, 1989 ACM 0-89791-306-X/89/0006/0110.
- [50] Koopman Phillip, J., Lee Peter, Siewiorek Daniel P. (1992). *Cache Behaviour of Combinator Graph Reduction*, ACM Transactions on programming languages and system Vol. 14, No.2, April 92, pp265-297.

- [51] Lins, Rafael D. (1987). *Categorical Multi-Combinators*, LNCS No.274, Functional programming languages and computer architecture, 1987.
- [52] MacLennan, Bruce J. (1990). **Functional Programming, Practice and Theory**, page 444, Addison-Wesley 1990.
- [53] Mannino, Michael V. (1990). *Object-Oriented paradigm*, IEEE Transcation on software engineering V-16, No.11, pp 1247, Nov 90.
- [54] McGregor, John D. and Korson, Tim (1990). *Object-oriented design*, Communication of ACM, V-33, No-9, pp 43, Sept 90.
- [55] Michaelson, Greg (1989) **An introduction to Functional programming through Lambda Calculus** Addison-Wesley, 1989.
- [56] Milne, Allan C. (1987). *Analysis and Manipulation of BNF definition*, EUUG Autumn Conference Proceedings, pp 105-122, EUUG, 1987.
- [57] Minitab Data Analysis Software (1988). *Minitab reference manual*, Release 6.1 January 1988. Minitab, Inc. 3081 Enterprise drive, State college, PA 16801, U. S. A.
- [58] Natanson L D, Samson W B and Wakelin A W (1990) *Objected-oriented implementations from a functional specification*, 1990, Dept. of Mathematical and Computer Sciences, Dundee Institute of Technology, Bell Street, Dundee DD1 1HG.
- [59] Neil, Graham (1983). **Introduction to Pascal**, Second Edition, 1983.
- [60] Reade, Chris (1989) **Elements of Functional Programming**, page 443, Addison-Wesley ,1989.
- [61] Richard B. Kieburtz (1985). *The G-machine: A fast, graph-reduction evaluator*, LNCS No. 201, Functional programming and computer architecture Nancy, France, Sept-1985.

- [62] Sebesta, Robert W. (1989). **Concepts of Programming Languages**, The Benjamin/Cumming Series in Computer Science, First Edition, 1989.
- [63] Thompson, S. and Lins, R. (1992). *The categorical multi-combinator machine: CMCM*, The Computer Journal Vol. 35, No.2, 1992.
- [64] Treleaven, P. C., Brownbridge, D. R. and Hopkins, R. P. (1982). *Data Driven and Demand driven Computer Architecture*, computing surveys, vol-14, No. 1, March 82, pp 93-143.
- [65] Turner, D. A. (1979). *A New Implementation Technique for applicative Languages*, Software Practice and Experience, Vol 9, pp 31-49, 1979.
- [66] Turner, D. A. (1985). *Miranda: A non strict functional language with polymorphic types*, Proceedings of the IFIP international conference on functional programming languages and computer architecture, 1985.
- [67] Turner, Raymond (1991) **Constructive foundations for functional languages**, McGraw-Hill 1991.
- [68] Vegdahl, Steven R. (1984). *A survey of Proposed Architecture for the Execution of Languages*, IEEE, Transactions on Computers, Dec-84, Vol C-33, No. 12, 1984.
- [69] Wegner, Peter (1986). *ACM SIGPLAN Notices*, Vol. 21, No. 10, 1986.
- [70] Wegner, P (1986) *Language paradigm for programming in the Large*, Annual lecture course 1985/86, (CS/86/3), University of St. Andrews, Department of Computational Science, 1986.
- [71] Weinreb D. and Moon D. (1981) *LISP machine manual*, Symbolics, 1981.
- [72] Wikstrom, Ake (1987) **Functional programming using standard ML**, Prentice Hall International (UK), 1987.

- [73] Wilson, L. B. and Clark, R. G. (1988). **Comparative programming languages**, Addison-Wesley, 1988.
- [74] Winter, S. C., Gupta, J. P. and Wilson, D. R. (1984). *Data and Demand driven computer architecture*, Advances in Microprocessing and Microprogramming, EUROMICRO, pp 287-296, 1984.
- [75] Wadsworth, C. P. (1971). *Semantics and pragmatics of the Lambda calculus*, D. Phil. Thesis, University of Oxford, 1971.
- [76] Wray, S. C. and Fairbairn, J. (1989). *Non-strict Languages Programming and Implementation*, The Computer Journal, pp 142-152, V-32, No. 2, 1989.
- [77] Abelson, H. and Sussman, G. J. (1986). **The structure and interpretation of computer programs**, MIT, Press, Cambridge, Mass.
- [78] Burstall, R. M., Collins, J. S. and Poppleston, R. J. (1971). **POP-2**, Edinburgh University Press (1971).
- [79] Kleene, S. C. (1936). *λ -definability and recursiveness*, Duke Math. J., pp 340-353, 1936.
- [80] Landin, P. (1965). *A correspondance between Algol60 and Church's lambda notation*, CACM, Vol. 8, pp 89-101, 1965.
- [81] Milner, R. (1978). *A theory of polymorphism in programming*, (gives details of language ML's polymorphism), CSR-9-77, Edinburgh University (1978). Also in J. Comp. Sys. Sciences, Vol. 17, pp 348-375 (1978).
- [82] Turner, D. A. (1976). **The SASL language manual**, University of St Andrews, 1976.

Appendices

APPENDIX - A

A λ -calculus Language Syntax (ALCAL) expressed in an
extended BNF notation [56].

$\langle \text{dialogue} \rangle$::=	$[\langle \text{instruction} \rangle \mid \langle \text{expression} \rangle]^*$;
$\langle \text{instruction} \rangle$::=	def $\langle \text{name} \rangle = \langle \text{expression} \rangle \mid$ exit. \mid load $\langle \text{string} \rangle .$
$\langle \text{expression} \rangle$::=	$\langle \text{function} \rangle \mid \langle \text{if} \rangle \mid \langle \text{list-expr} \rangle .$
$\langle \text{function} \rangle$::=	lam $\langle \text{name} \rangle \text{ " " } \langle \text{expression} \rangle .$
$\langle \text{if} \rangle$::=	if $\langle \text{expression} \rangle$ then $\langle \text{expression} \rangle$ else $\langle \text{expression} \rangle .$
$\langle \text{list-expr} \rangle$::=	$\langle \text{boolean-expr} \rangle \{ :: \langle \text{boolean-expr} \rangle \}^* .$
$\langle \text{boolean-expr} \rangle$::=	$\langle \text{boolean-term} \rangle \{ [\text{and} \mid \text{or}] \langle \text{boolean-term} \rangle \}^* .$
$\langle \text{boolean-term} \rangle$::=	{not} $\langle \text{relation-expr} \rangle .$
$\langle \text{relation-expr} \rangle$::=	$\langle \text{arith-expr} \rangle \{ \langle \text{rel-op} \rangle \langle \text{arith-expr} \rangle \} .$
$\langle \text{arith-expr} \rangle$::=	$\langle \text{arith-term} \rangle \{ [+ \mid -] \langle \text{arith-term} \rangle \}^* .$
$\langle \text{arith-term} \rangle$::=	$\langle \text{factor} \rangle \{ [" * " \mid \text{div} \mid \text{mod}] \langle \text{factor} \rangle \}^* .$
$\langle \text{factor} \rangle$::=	$\langle \text{item} \rangle \{ " " \langle \text{item} \rangle \}^* .$
$\langle \text{item} \rangle$::=	$\langle \text{name} \rangle \mid \langle \text{constant} \rangle \mid (\langle \text{expression} \rangle) \mid$ $\langle \text{list} \rangle .$
$\langle \text{constant} \rangle$::=	$\langle \text{integer} \rangle \mid \langle \text{boolean} \rangle \mid \langle \text{nil-list} \rangle \mid$ $\langle \text{function-const} \rangle .$
$\langle \text{list} \rangle$::=	$"[" \{ \langle \text{expression} \rangle \{ , \langle \text{expression} \rangle \}^* \} "]" .$
$\langle \text{boolean} \rangle$::=	true \mid false .
$\langle \text{nil-list} \rangle$::=	nil .
$\langle \text{function-const} \rangle$::=	hd \mid tl .
$\langle \text{rel-op} \rangle$::=	$"<" \mid "<=" \mid ">" \mid ">=" \mid "=" \mid "!=" .$

microsyntax

$\langle \text{name} \rangle ::= [\langle \text{letter} \rangle]^* .$

$\langle integer \rangle ::= \{ - \} [\langle digit \rangle]^*.$
 $\langle letter \rangle ::= \text{"any upper or lower case alphabetic character"}.$
 $\langle digit \rangle ::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9.$

APPENDIX - B

B1.1: Lingo code for interpreter (Alcal)

```
/* Alcal interpreter : myk   Sat Sep 28 10:07:07 BST 1991  */

Alcal is Object
[scanner recovering symbolTable parse code count ]
{
new []
{^(((super new) scanner:( Scanner
  keywords:(Vector [ "def" "show" "if" "then" "else" "mod" "lam"
    "load" "div" "and" "or" "not" "exit" "hd" "tl" "true" "false" "nil" ])
  punctuationCharacters:( Vector [ ',' '.' '(' ')' '[' ']' ';' '+' '-' '/' ])
  crypticCharacters:(Vector ['=' '<' '>' '*' ':' ])
  source:(FileDescriptor input))) start);}.}

close []
{ /* to close resulting code, count file */
  code close; count close;}.

self mustBe: aToken []
{ /* this method is used when there is no option in the choice
  of production */
  if recovering
  then {while (aToken != (scanner token)) and (scanner token != ";")
    do {scanner getToken;}
    recovering := Boolean false;}
  else {if aToken = (scanner token)
    then {scanner getToken;}
    else {"syntax error \n" printedOn: FileDescriptor output;
      self error:aToken;}}}.

self have: aToken []
{ /* this method gives a choice between productions */
  if aToken = (scanner token)
  then {scanner getToken;
    ^Boolean true;}
  else {^Boolean false;}}.
```

```

self error:aToken [ expected ]
{ /* for an error (unexpected symbol in the input stream), a syntax
    error is reported to display a specific message */
  if (Vector [ "false" ] includes: aToken )
  then expected := "expression"
  else expected := aToken;
    "\nexpected " printedOn: FileDescriptor output;
    expected printedOn: FileDescriptor output with:" found ";
    (scanner token) printedOn: FileDescriptor output with:"'.\n";
    "\n expected " printedOn:parse;
    recovering := Boolean true;}.

```

```

self start []
{ /* Main interpretation method */
  symbolTable := Dictionary [];
  recovering := Boolean false;
  "\nFL> " printedOn: FileDescriptor output;
  scanner getToken;
  while (scanner token) != "exit" do
  {while (scanner token) != ";" do {self dialogue; }
  {"\nFL> " printedOn: FileDescriptor output;
  recovering := Boolean false; scanner getToken;}}
  "\nExiting FL\n " printedOn: FileDescriptor output;
  self close;}.

```

```

self dialogue [startTime finishTime elapsedTime ocStart ocFinish result
noofobjectso noofobjectsn]
{ /* Main dialogue, for detail of rules see appendix - A, each
    of these rule contains alternatives, and each alternative
    is inspected to determine which alternate rule should be
    followed. */
  if (scanner token) = "load"
  then {self load;}
  else
    if (scanner token ) = "def"
    then {self instruction; }
    else {
      result      := self expression ;
      Object gc;
      "\n<gc activated>\n" printedOn:FileDescriptor output;
      ocStart      := Object objectsInCore;
    }

```

```

        noofobjectso := Object objectCount;
        Time now;
        result      := result evaluate;
        startTime   := Time time;
        Time now;
        finishTime  := Time time;
        ocFinish    := Object objectsInCore;
        noofobjectsn := Object objectCount;
        elapsedTime := finishTime - startTime;
        result printedOn: FileDescriptor output;
        " calcualted in " printedOn: FileDescriptor output;
        elapsedTime printedOn: FileDescriptor output with:" secs\n";
        "number of objects = " printedOn: FileDescriptor output;
        (noofobjectsn - noofobjectso) printedOn: FileDescriptor output
        with:"\n";}}.

self load [nextToken theOldFile filename fileDescriptor ]
{ /* an auxiliary function to scan all the function definitions in
    a file */
    self mustBe:"load";
    self mustBe:"String";
    filename := scanner theLastString;
    nextToken := scanner token;
    theOldFile := scanner source;
    "\n> " printedOn:FileDescriptor output with:filename;
    "<\n" printedOn:FileDescriptor output;
    fileDescriptor := FileDescriptor openForReading:((MetaClass
        archiveDirectory)+filename);
    scanner source: fileDescriptor;
    self start1;
    scanner source: theOldFile;
    scanner token:nextToken;
    fileDescriptor close; }.

self start1 []
{ /* this method detects the end-of-file of the input file opened
    by the previous method (load). */
    scanner getToken;
    while ((scanner token) != "exit") do
        {while (scanner token ) != ";" do {self dialogue;}
        {recovering := Boolean false;scanner getToken;}}}.

self instruction [ theName theTree dmy ]
{self mustBe:"def" ;

```

```

theName := self name;
dmy := Dummy new;
dmy theName:(theName) ;
symbolTable at:(theName symbol) put:dmy;
self mustBe:"=";
theTree := self expression;
dmy theTree:theTree ;}.

self expression [ch result]
{if (scanner token) = "lam"
 then {result := self function; ^result;}
 else {if (scanner token) = "if"
        then {result := self iff; ^result;}
        else {result := self list_expr; ^result;}}}.

self name [result]
{self mustBe:"Identifier";
 result := Variable named:(scanner theLastIdentifier); ^result;}.

self function [ var result exp1]
{self mustBe:"lam";
 var := (self name) symbol;
 self mustBe:".";
 exp1 := self expression ;
 result := exp1 abstract:var;
 ^ result;}.

self iff [result]
{
 {self mustBe:"if";
  result := Application of:Cond new to:(self expression);
  self mustBe:"then";
  result := Application of:result to:(self expression);
  self mustBe:"else";
  result := Application of:result to:(self expression);}
 ^result;}.

self list_expr [result a b]
{if (self have:"nil")
 then {result := (Constant value:(List new)); ^result;}
 else {a := self boolean_expr;
       if (self have:"::")
       then {b := self list_expr;
             result := Application of:(Application of:(Cons new)

```

```

        to:a) to:b;}
    else {result := a;}
    ^result;}}.

self boolean_expr [result]
{result := self boolean_term;
 while (scanner token) = "and" or (scanner token) = "or" do
 {if self have:"and"
  then {result := Application of:(Application of:And new to:
      result) to:(self boolean_term);}
  else {self mustBe:"or";
       result := Application of:(Application of:Or new to:
      result) to:(self boolean_term);}}
 ^result;}.

self boolean_term [result]
{if (scanner token) = "not"
 then {self mustBe:"not";
      result := Application of:Not new to:(self relation_expr);}
 else {result := self relation_expr; }
 ^ result;}.

self relation_expr [result]
{result := self arith_expr;
 while (Vector ["<" "<=" ">" ">=" "=" "~="] includes:(scanner token)) do
 {if self have:"<"
  then {result := Application of:(Application of:Lessthen new to:
      result) to:(self arith_expr);}
  else if self have:"<="
   then {result := Application of:(Application of: Lessthenequal new
      to:result) to:(self arith_expr);}
  else if self have:">"
   then {result := Application of:(Application of: Greaterthen
      new to:result) to:(self arith_expr);}
  else if self have:">="
   then {result := Application of:(Application of:
      Greaterthenequal new to:result) to:
      (self arith_expr);}
   else if self have:"="
   then {result :=Application of:(Application of:Equal
      new to:result) to:(self arith_expr);}
  else if self have:"~=" do
   {result := Application of:(Application of:
      Notequal new to:result) to:

```

```

                                (self arith_expr);}}
                                ^result;}}.

self arith_expr [result]
{result := self arith_term;
 while (scanner token) = "+" or (scanner token) = "-" do
 {if   self have:"+"
  then {result := Application of:(Application of:Plus new to:result)
        to:(self arith_term);}
  else {self mustBe:"-";
        result := Application of:(Application of:Minus new to:result)
        to:(self arith_term);}}
  ^result;}}.

self arith_term [result]
{result := self factor;
 while (Vector ["*" "div" "mod"] includes:(scanner token)) do
 {if   self have:"*"
  then {result := Application of:(Application of:Times new
        to:result) to:(self factor);}
  else if self have:"div"
  then {result := Application of:(Application of:Divide new
        to:result) to:(self factor);}
  else if self have:"mod" do
    {result := Application of:(Application of:(Mod new) to:
      result) to:(self factor); }}
  ^result;}}.

self factor [result]
{result := self item;
 while (Vector ["Identifier" "Integer" "true" "false" "nil" "hd"
"t1" "[" "(" ] includes:(scanner token)) do
 {result := Application of:result to:(self item);}
 ^result;}}.

self item [result]
{if (Vector ["Identifier" "("] includes:(scanner token))
 then {if self have:"Identifier"
  then {if symbolTable includes:(scanner theLastIdentifier)
    then {result := (symbolTable at:(scanner theLastIdentifier));
    ^result;}}
    else {result := Variable named:(scanner theLastIdentifier);
    ^result;}}
  else if self have:"(" do

```

```

        {result := self expression; self mustBe:"");}
        ^result;}
else {if self have:"["
    then {result := self list;}
    else {result := self constant;}
    ^result;}}.

self list [a b result]
{if ((scanner token) = "]") not
    then {a := self expression;
        if self have:", "
        then {b:= self list;
            result := Application of:(Application of:(Cons new)
                to:a) to:b;}
        else {self mustBe:""];
            result := Application of:(Application of:(Cons new)
                to:a) to:(Constant value:(List new));}}
    else {self mustBe:""];
        result := Constant value:(List new);}
    ^result;}.

self constant [result]
{if self have:"Integer"
    then {result := Constant value:(scanner theLastInteger);^result;}
    else if self have:"nil"
        then {result := Constant value:(List new);^result;}
    else if self have:"hd"
        then {result := Application of:(Hd new) to:(self item);
            ^result;}
        else if self have:"tl"
            then {result := Application of:(Tl new) to:(self item);
                ^result;}
        else if self have:"true"
            then {^Constant value:(Boolean true);}
            else {self mustBe:"false";
                ^Constant value:(Boolean false);}}.
scanner: a [] { scanner := a }.

```


B1.2: Lingo code for various combinators

Each combinator module have their own instance method called 'evaluate' while they all inherits the instance method 'abstract' from the module '*Operator*'.

The module hierarchy representing various combinator objects, various type of node and leaves are defined as:

$$Plus \subset Operator \subset Node \subset Object$$

Node is Object

```
[symbol]
{}
symbol:aString []
{symbol := aString;}.
symbol [] { ^ symbol; }.
printedOn:f []
{symbol printedOn:f;}.
```

Operator is Node

```
[]
{ }
self abstract:aString [result]
{result := Application of:(K new) to:self;^result;}.

```

Variable is Node

```
[]
{named:aString [] {^((super new) symbol:aString);}.}
self abstract:aString []
{if (self symbol) = aString
then ^(I new)
else ^(Application of:(K new) to:self);}.

```

Constant is Node

```
[]
{value: aValue [] {^ ((super new) symbol: aValue);}.}
self abstract:aString []
{^Application of:(K new) to:self;}.
self evaluate [ ] { ^ (self symbol); }.

```

Lambda is Node

```
[bv exp]
{new:avar in:ex [ temp ]

```

```

{temp := super new;
 temp symbol:("lam " + avar + ". ");
 temp bv:avar;
 temp exp:ex;
 ^temp}.
}
self abstract:aString [temp result]
 {temp := exp abstract:bv; ^temp;}.
self printedOn:f []
 {(self symbol) printedOn:f; exp printedOn:f;}.
bv:b []{bv:=b;}.
exp:c []{exp:=c;}.

```

```

Application is Node
[left right]
{of:alpha to:beta []
 ^((super new) left:alpha right:beta);}.
}
self abstract:aString [a b resulta result]
{a      := left  abstract:aString;
 b      := right abstract:aString;
 resulta := Application of:(S new) to:a;
 result  := Application of:resulta to:b;
 ^result};}.
evaluate [] { ^ (( left evaluate ) performWith:(right )); }.
printedOn:f []
 {"(" printedOn:f; left  printedOn:f; " "  printedOn:f;
  right printedOn:f; ")" printedOn:f; }.
left:a right:b []
 {left  := a; right := b;}.
left [] {^left}.
right [] {^right}.

```

```

S is Operator
[]
{new [] { ^((super new) symbol:"S");}.}
self evaluate []
 {^module f []
  {^(module g of f []
    {^(module x of g f []
      {^(Application of:(Application of:f to:x)
        to:(Application of:g to:x)) evaluate;}})}}}.

```

K is Operator

```
[]  
{new [] {^((super new) symbol:"K");}.}  
self evaluate []  
  {^module x []  
    {^(module y of x []  
      {^(x evaluate);})}}.
```

I is Operator

```
[]  
{new [] {^((super new) symbol:"I");}. }  
self evaluate []  
  {^module x []  
    {^(x evaluate);}}.
```

B is Operator

```
[]  
{new [] { ^((super new) symbol:"B");}.}  
self evaluate []  
  {^module f []  
    {^(module g of f []  
      {^(module x of g f []  
        {^(Application of:f to:(Application of:g to:x))evaluate;})}})}.
```

C is Operator

```
[]  
{new [] { ^((super new) symbol:"C");}.}  
self evaluate []  
  {^module f []  
    {^(module g of f []  
      {^(module x of g f []  
        {^(Application of:(Application of:f to:x) to:g)evaluate;})}})}.
```

Sprime is Operator

```
[]  
{new [] { ^((super new) symbol:"S'");}.}  
self evaluate []  
  {^module k []  
    {^(module f of k []  
      {^(module g of f k []  
        {^(module x of g f k []  
          {^(Application of:(Application of:k to:(Application of:f  
            to: x)) to:(Application of:g to:x)) evaluate;})}})}.
```

Cprime is Operator

```
[]  
{new [] { ^((super new) symbol:"C'");}.}  
self evaluate []  
  {^module k []  
    {^(module f of k []  
      {^(module g of f k []  
        {^(module x of g f k []  
          {^(Application of:(Application of:k to:(Application  
of:f to:x)) to:g) evaluate;}}))}}}.  
}
```

Bprime is Operator

```
[]  
{new [] { ^((super new) symbol:"B'");}.}  
self evaluate []  
  {^module k []  
    {^(module f of k []  
      {^(module g of f k []  
        {^(module x of g f k []  
          {^(Application of:(Application of:(k to:f) to:  
Application of:g to:x)) evaluate;}}))}}}.  
}
```

Bstar is Operator

```
[]  
{new [] { ^((super new) symbol:"B*"); }.}  
self evaluate []  
  {^module k []  
    {^(module f of k []  
      {^(module g of f k []  
        {^(module x of g f k []  
          {^(Application of:k to:(Application of:f to:(  
Application of:g to:x))) evaluate;}}))}}}.  
}
```

Cond is Operator

```
[]  
{new [] { ^((super new) symbol:"cond");}.}  
self evaluate []  
  {^module x[]  
    {^(module y of x []  
      {^(module z of y x []  
        {if (x evaluate)  
then {^(y evaluate);}  
else {^(z evaluate);};}}))}}}.  
}
```

Mod is Operator

```
[]  
{new [] {^((super new) symbol:"mod");}.}  
self abstract:aString []  
  {^Application of:(K new) to: self;}.  
self evaluate []  
  {^module x [] {^(module y of x [] {^((x evaluate) % (y evaluate))))}}.
```

Few more objects for Lists (for example *Cons*, *Hd*, *Tl*, *Nil*, *And*, *Or*, *Not*) and for primitive arithmetic operators like *Plus*, *Minus* etc are defined as:

Cons is Operator

```
[]  
{new [] { ^((super new) symbol:"::");}.}  
self abstract:aString []  
  {^Application of:(K new) to:self; }.  
self evaluate []  
  {^module y [ ] {^(module x of y [xList a ]  
    {a := List new;  
      a append:(y evaluate);  
      xList := x evaluate;  
      until xList isEmpty do  
        {a append:(xList first); xList := xList next;}  
        ^a;}})}}.
```

Hd is Operator

```
[]  
{new [] { ^((super new) symbol:"hd");}.}  
self abstract:aString []  
  {^Application of:(K new) to:self;}.  
self evaluate []  
  {^module x [] {^((x evaluate) first)}}.
```

Tl is Operator

```
[]  
{new [] { ^((super new) symbol:"tl");}.}  
self abstract:aString []  
  {^Application of:(K new) to:self;}.  
self evaluate []  
  {^module x [] {^((x evaluate) next)) } }.
```

Not is Operator

```
[]  
{new [] { ^((super new) symbol:"not");}.}  
self abstract:aString []  
  {^Application of:(K new) to:self;}.  
self evaluate []  
{^module x [] {^((x evaluate) not);}}.
```

Plus is Operator

```
[]  
{new [] { ^((super new) symbol:"+");}.}  
self abstract:aString []  
  {^Application of:(K new) to:self;}.  
self evaluate []  
  {^module x [] {^(module y of x [] {^((x evaluate)+(y evaluate));}})}.
```

Note:

Objects like *Minus*, *Divide*, *Times*, *Lessthen*, *Equal*, *NotEqual* *Not*, *And*, *Or* are similar to *Plus*.

Dummy is Object

```
[theTree theName ]  
{ }  
theTree: aTree [] {theTree := aTree }.  
theName: aName [] {theName := aName }.  
kind [] { ^ "Dummy" }.  
theTree [] {^theTree}.  
self abstract:aString [] {^Application of:K to:self;}.  
self evaluate [] {^(theTree evaluate);}.  
theName [] {^theName}.  
printedOn:f [] {theName printedOn: f;}.
```

B1.3: Smalltalk-80 code for various combinators in *Fact 3*.

```
Smalltalk at:#z    put: 0 !
Smalltalk at:#a1   put: 0 !
Smalltalk at:#a2   put: 0 !
Smalltalk at:#dmy  put: 0 !

Object subclass: #Node
    instanceVariableNames: 'symbol'
    classVariableNames: ''
    poolDictionaries: ''
    category: nil !
!Node methodsFor: 'testing'!
symbol: aString
symbol := aString.
!
symbol
^ symbol
!
print
symbol print.
!
printNl
symbol printNl.
!!

Node subclass: #Operator
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''
    category: nil !

Node subclass: #Application
    instanceVariableNames: 'left right'
    classVariableNames: ''
    poolDictionaries: ''
    category: nil !
!Application class methodsFor: 'instance creation'!
of: alpha to: beta
    ^((super new) left: alpha right: beta)
!!
!Application methodsFor: 'testing'!
left: a
    left := a.
```

```

!
right: b
    right := b.
!
left: a right: b
    self left: a.
    self right: b.
!
evaluate
^ (( left evaluate) value: (right))
!
print
$( print.
left print.
$ print.
right print.
$) print.
!
printNl
$( print.
left print.
$ print.
right print.
$) printNl.
!!

Operator subclass: #Minus
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''
    category: nil !
!Minus class methodsFor: 'instance creation'!
new
^ ((super new) symbol: $-)
!
evaluate
^ [ :x | [ :y | ((x evaluate) - (y evaluate)) ] ]
!!

Operator subclass: #Times
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''
    category: nil !

```



```

!Times class methodsFor: 'instance creation'!
new
    ^((super new) symbol: $*)
!
evaluate
    ^ [ :x | [ :y | ((x evaluate) * (y evaluate)) ] ]
!!

Operator subclass: #Equal
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''
    category: nil !
!Equal class methodsFor: 'instance creation'!
new
    ^ ((super new) symbol: $=)
!
evaluate
    ^ [ :x | [ :y | ((x evaluate) = (y evaluate)) ] ]
!!

Node subclass: #Constant
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''
    category: nil !
!Constant class methodsFor: 'instance creation'!
value: aValue
    ^ ((super new) symbol: aValue)
!!
!Constant methodsFor: 'testing'!
evaluate
    ^ (self symbol)
!!

Operator subclass: #Cond
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''
    category: nil !
!Cond class methodsFor: 'instance creation'!
new
    ^((super new) symbol: 'Cond' )
!

```

```

evaluate
^ [ :x | [ :y | [ :z | (x evaluate)
    ifTrue: [ ( y evaluate)]
    ifFalse: [ ( z evaluate)] ]]]
!!

```

```

Object subclass: #Dummy
    instanceVariableNames: 'theTree theName'
    classVariableNames: ''
    poolDictionaries: ''
    category: nil !
!Dummy    methodsFor: 'testing'!
theTree: aTree
    theTree := aTree.
!
theName: aName
    theName := aName.
!
kind
    'Dummy'.
!
theTree
    ^ (theTree)
!
theName
    ^ (theName)
!
evaluate
    ^ (theTree evaluate)
!
printOn: aFile
    theName printOn: aFile.
!!

```

```

Operator subclass: #Factsc1
    instanceVariableNames: ''
    classVariableNames: 'dmy'
    poolDictionaries: ''
    category: nil !
!Factsc1 class methodsFor: 'instance creation'!
new
    ^ ((super new) symbol: 'Factsc1')
!

```

```

    evaluate
      ^ ( dmy )
!
initialise | a1 |
  dmy := Dummy new.
  dmy theName: ('fact').
  a1 :=
    [ :x |
      ((Application of: (Application of: (Application of:
        Cond to: (Application of: (Application of: Equal to: x)
          to: (Constant value: 0))) to: (Constant value: 1))
        to: ( Application of: (Application of: Times to: x)
          to: ( Application of: (Factsc1 initialise) to: (
            Application of: (Application of: Minus to: x)
              to: (Constant value: 1)) )))evaluate )].
  dmy := a1
!
  dmy
  ^ ( dmy )
!!

Operator subclass: #Fact
  instanceVariableNames: ''
  classVariableNames: ''
  poolDictionaries: ''
  category: nil !
!Fact      class methodsFor: 'instance creation'!
test: x
  ^((Application of: (Factsc1 initialise) to:
    (Constant value: x)) evaluate)
!!

```

APPENDIX - C

C1.1: Some representative functions (Bench-marks)

In order to measure the performance of the various implementations of a functional language a number of benchmark programs have been proposed. These are designed to measure and/or count such features as:

- execution time
- number of objects created
- activations of garbage collection.

Suitable benchmarks include

- functions with one argument
- functions with more than one argument
- simple recursive function
- functions involving nested recursion

A selection of benchmarks must attempt to cover the range from the simplest functions to the most complex ones.

The benchmarks used used for the purpose of this study are:

- Ex0 — a simple function with one argument
- Ex1 — a simple function with several arguments
- Fact — a (factorial) tail-recursive function
- Ack — a (ackermann) function involving nested double recursion.

These are defined below in the Alcal syntax:

1. *Ex0* It is a simple succ function with one variable. It is represented in Alcal as:

```
def f = lam x. x + 1;
```

for example

```
f 4 = 5
```

2. *Ex1* It is a function of nested λ -terms, it includes multivariable (bound and free) .

```
def g = (lam x.( lam y.(x + (lam x.(x - 3)) y)) 5) 6;
```

for example

```
g 2 = 8
```

3. *Fact* The function factorial(n) is classical bench mark.

```
def fact = lam x. if x = 0 then 1 else x * fact(x - 1);
```

for example

```
fact 5 = 120
```

4. *Ack* The ackermann function grows exceedingly quickly. The function uses double recursion with one recursive application nested within the other. In Alca it may be defined as:

```
def ack = lam x. lam n.  
  if x = 0  
  then n + 1  
  else if n = 0  
    then ack(x-1) 1  
    else ack(x-1)(ack x (n-1));
```

for example

```
ack 3 6 = 509
```

C1.2: Benchmark performance

The four benchmark programs defined above were run many times over within a test harness program and the following data was collected during these runs.

- number of times benchmark program was executed
- total elapsed time
- number of objects created
- number of times garbage collection was activated
- number of times benchmark program executed before the garbage collection was first activated.
- time taken for test harness program without benchmark (empty loop)

C1.3: Intermediate code for λ -lifting and SKI-implementation

1. Ex0 (succ)

Alcal's representation :

$def\ a = lam\ x.\ x + 1;$

Intermediate form for λ -lifting:

$((+ x)\ 1)$

Intermediate form for SKI-combinator:

$((C\ +)\ 1)$

eg. $a\ 4 \implies 5$

2. Ex1

Alcal's representation :

$def\ a = lam\ x.\ (lam\ y.\ (x+(lam\ x.\ (x-3))\ y))\ 5\ ;$

Intermediate form for λ -lifting:

$((+ x)\ (((- x)\ 3)\ y))$

Intermediate form for SKI-combinator:

$((C\ (((C'\ B)\ +)\ ((C\ -)\ 3)))\ 5)$

eg. $a\ 6 \implies 8$

3. Fact

Alcal's representation :

*def fact = lam x. if x = 0 then 1 else x * fact(x-1);*

Intermediate form for λ -lifting:

((cond ((= x) 0)) 1) ((x) (fact ((- x) 1))))*

Intermediate form for SKI-combinator:

*((S (((C' cond) ((C =) 0)) 1)) ((S *) ((B fact) ((C -) 1))))*

eg. fact 3 \implies 6

4. Ack

Alcal's representation :

*def ack = lam x. lam n. if x = 0 then n+1 else if n = 0
then ack(x-1) 1 else ack(x-1) (ack x(n-1));*

Intermediate form for λ -lifting:

*((cond ((= x) 0)) ((+ n) 1)) (((cond ((= n) 0)) ((ack ((- x) 1))
1)) ((ack ((- x) 1)) ((ack x) ((- n) 1)))))*

Intermediate form for SKI-combinator:

*((S (((C' S') ((B cond) ((C =) 0)) ((C +) 1))) (((S' S) ((B ((C'
cond) ((C =) 0)) (((C' ack) ((C -) 1)) 1))) (((S' B) ((B ack) ((C
-) 1))) (((C' B) ack) ((C -) 1)))))*

eg. ack 1 1 \implies 3

Some other functions

Intermediate forms for some other functions are also given which includes list constructor ':: nil ', hd (represents head of a list), tl (tail of a list) boolean function true and false .

5. fib

Alcal's representation :

$\text{def fib} = \text{lam } n. \text{ if } n < 2 \text{ then } 1 \text{ else fib}(n-1)+\text{fib}(n-2);$

Intermediate form for λ -lifting:

$((\text{cond } ((< \ n) \ 2)) \ 1) \ ((+ \ (\text{fib } ((- \ n) \ 1))) \ (\text{fib } ((- \ n) \ 2))))$

Intermediate form for SKI-combinator:

$((S \ (((C' \ \text{cond}) \ ((C \ <) \ 2)) \ 1)) \ (((S' \ +) \ ((B \ \text{fib}) \ ((C \ -) \ 1))) \ ((B \ \text{fib}) \ ((C \ -) \ 2))))$

eg. $\text{fib } 12 \implies 233$

6. length

Alcal's representation :

$\text{def length} = \text{lam } x. \text{ if } x = \text{nil} \text{ then } 0 \text{ else } 1+\text{length}(\text{tl } x);$

Intermediate form for λ -lifting:

$((\text{cond } ((= \ x) \ \text{nil})) \ 0) \ ((+ \ 1) \ (\text{length } (\text{tl } x))))$

Intermediate form for SKI-combinator:

$((S \ (((C' \ \text{cond}) \ ((C \ =) \ \text{nil})) \ 0)) \ ((B \ (+ \ 1)) \ ((B \ \text{length}) \ \text{tl})))$

eg. $\text{length } [1, 9, 10, 11, 13, 5] \implies 6$

7. append

Alcal's representation :

```
def append = lam a. lam x . if x = nil then a :: nil
else hd x::append a(tl x);
```

Intermediate form for λ -lifting:

```
((cond ((= x) nil)) ((:: a) nil)) ((:: (hd x)) (append a) (tl x)))
```

Intermediate form for SKI-combinator:

```
((S' S) (B ((C' cond) ((C =) nil))) ((C ::) nil)) (B ((S' ::) hd))
((C' B) append) tl)))
```

eg. *append* 1 [2, 3, 4] \Longrightarrow [2, 3, 4, 1]

8. concat

Alcal's representation :

```
def concat = lam x. lam y. if x = nil then y else hd x :: concat(tl x)y;
```

Intermediate form for λ -lifting:

```
((cond ((= x) nil)) y) ((:: (hd x)) (concat (tl x)) y)))
```

Intermediate form for SKI-combinator:

```
((S' S) (B cond) ((C =) nil)) ((S' B) ((B ::) hd)) (B concat)
tl)))
```

eg. *concat* [1, 2, 3][4, 5] \Longrightarrow [1, 2, 3, 4, 5]

eg. *concat* [[1, 2] :: [3] :: nil] [7] \Longrightarrow [[[1, 2], [3]], 7]

9. map

Alcal's representation :

```
def map = lam f. lam l. if l = nil then nil else f (hd l):: map f(tl l);
```

Intermediate form for λ -lifting:

```
((cond ((= l) nil)) nil) ((:: (f (hd l))) ((map f) (tl l))))
```

Intermediate form for SKI-combinator:

```
((B (S (((C' cond) ((C =) nil)) nil))) (((S' (S' ::)) ((C B) hd))  
(((C' B) map) tl)))
```

Similarly 'twice' and 'maptw' can be defined as:

```
def twice = lam x. 2*x;
```

```
def maptw = map twice;
```

eg. map twice [1, 2, 3] \Rightarrow [2, 4, 6]

eg. maptw [1, 2, 3] \Rightarrow [2, 4, 6]

10. fold

Alcal's representation :

```
def fold = lam f. lam i. lam l. if l = nil then i  
else f (hd l) (fold f i (tl l));
```

Intermediate form for λ -lifting:

```
((cond ((= l) nil)) i) ((f (hd l)) (((fold f) i) (tl l))))
```

Intermediate form for SKI-combinator:

```
((B ((S' S) ((C' cond) ((C =) nil)))) (((S' B) ((C S') hd)) (((C'  
(C' B)) fold) tl)))
```

Similarly 'add' can be defined as:

```
def add = lam x. lam y. x + y;
```

eg. fold add 0 [3, 7, 2] \Rightarrow 12

eg. fold add 5 [3, 7, 2] \Rightarrow 17

eg. fold append [1] [2, 3] \Rightarrow [1, 1, 1]

eg. fold append [] [1, 2, 3] \Rightarrow nil

11. sumlist

Alcal's representation :

def sumlist = lam x. if x = nil then 0 else hd x + sumlist (tl x);

Intermediate form for λ -lifting:

((cond ((= x) nil)) 0) ((+ (hd x)) (sumlist (tl x)))

Intermediate form for SKI-combinator:

((S (((C' cond) ((C =) nil)) 0)) ((S' +) hd) ((B sumlist) tl)))

eg. sumlist [1, 2, 3, 4] \Rightarrow 10

12. member

Alcal's representation :

def member = lam l. lam i. if l = nil then false

else if (hd l) = i then true else member (tl l) i;

Intermediate form for λ -lifting:

((cond ((= l) nil)) aBoolean = false) (((cond ((= (hd l)) i)) aBoolean = true) ((member (tl l)) i)))

Intermediate form for SKI-combinator:

((S' B) (((C' cond) ((C =) nil)) aBoolean = false)) ((S' S) (((C' (C' cond)) ((B =) hd)) aBoolean = true) ((B member) tl)))

eg. member [1, 2, 3] 3 \Rightarrow Boolean true

eg. member [1, 2, 3] 4 \Rightarrow Boolean false

APPENDIX - D

D1.1: No. of reductions and No. of objects created per run.

<i>Comb. Object</i>	<i>Ex0</i>				<i>Ex1</i>				<i>Fact 3</i>				<i>Ack 1 1</i>			
	<i>Ins</i>		<i>cls</i>		<i>Ins</i>		<i>cls</i>		<i>Ins</i>		<i>cls</i>		<i>Ins</i>		<i>cls</i>	
	S	λ	S	λ	S	λ	S	λ	S	λ	S	λ	S	λ	S	λ
<i>Plus</i>	1	1	1	1	1	1	1	1	0	0	0	0	2	2	2	2
<i>Minus</i>	0	0	0	0	1	1	1	1	9	9	9	9	3	3	3	3
<i>Times</i>	0	0	0	0	0	0	0	0	3	3	3	3	0	0	0	0
<i>Equal</i>	0	0	0	0	0	0	0	0	4	4	4	4	6	6	6	6
<i>Cond</i>	0	0	0	0	0	0	0	0	4	4	4	4	6	6	6	6
<i>S</i>	0	0	0	0	0	0	0	0	7	0	7	0	6	0	6	0
<i>B</i>	0	0	0	0	1	0	1	0	7	0	7	0	9	0	9	0
<i>C</i>	1	0	1	0	2	0	2	0	17	0	17	0	11	0	11	0
<i>Sprime</i>	0	0	0	0	0	0	0	0	0	0	0	0	7	0	7	0
<i>Cprime</i>	0	0	0	0	1	0	1	0	0	0	0	0	8	0	8	0
<i>Dummy</i>	0	0	0	0	0	0	0	0	3	0	3	0	3	0	3	0
<i>Ex1sc1</i>	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0
<i>Ex1sc2</i>	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0
<i>Factsc1</i>	0	0	0	0	0	0	0	0	0	4	0	4	0	0	0	0
<i>Acksc1</i>	0	0	0	0	0	0	0	0	0	0	0	0	0	4	0	4
<i>Acksc2</i>	0	0	0	0	0	0	0	0	0	0	0	0	0	4	0	4
<i>Constant</i>	2	2	2	2	3	3	3	3	21	21	21	21	18	18	18	18
<i>Application</i>	5	2	5	2	17	7	17	7	137	48	137	48	178	49	178	49
<i>Total reduc</i>	9	5	9	5	26	14	26	14	212	93	212	93	257	89	257	89
<i>App O.creat</i>	5	2	5	2	17	7	17	7	84	41	84	41	149	97	149	97
<i>other O.cre</i>	4	3	2	2	9	7	3	3	16	39	5	13	38	68	10	28
<i>Total O.cre</i>	9	5	7	4	26	14	20	10	100	80	89	54	187	165	159	125

D1.2: Tables of timings on the Rekursiv and RISC architectures

<i>Combinator objects reduction time</i>	<i>Rekursiv (μ-sec)</i>	<i>RISC (μ-sec)</i>
<i>S, B, C, Cond, Sprime, Cprime, Ex1sc1, Ex1sc2, Factsc1, Acksc1, Acksc2, Plus, Minus, Times, Equal</i>	13.8	192.6
<i>Constant</i>	25.7	129.6
<i>Application</i>	31.0	245.2

Reductions time for various combinator objects on the Rekursiv and RISC architectures.

<i>Combinator objects object creation time</i>	<i>Rekursiv (μ-sec)</i>	<i>RISC (μ-sec)</i>
<i>S, B, C, Cond, Sprime, Cprime, Ex1sc1, Ex1sc2, Factsc1, Acksc1, Acksc2, Plus, Minus, Times, Equal, Constant</i>	31.46	280.0
<i>Application</i>	31.0	548.6

Objects creation time for various combinator objects on the Rekursiv and RISC architectures.

primitive arithmetic operator <i>executions time</i>	<i>Rekursiv</i> (μ -sec)	<i>RISC</i> (μ -sec)
+, -, =	2.11	15.2
*	7.85	15.2

Executions time of primitive arithmetic operators on the Rekursiv architecture (using Lingo language) and RISC architecture (using Smalltalk-80).

resource allocation <i>time</i>	<i>Rekursiv</i> (milli-sec)	<i>RISC</i> (milli-sec)
<i>garbage collection (gc)</i>	655	29

Time spent on garbage collecting on the Rekursiv and RISC architecture.

D1.3: Tables for combinator reductions for benchmarks.

<i>combinator objects</i>	<i>SKI-combinator implementation</i>	<i>Lambda-lifting implementation</i>
<i>Plus</i>	1	1
<i>Constant</i>	2	2
<i>C</i>	1	0
<i>Application</i>	5	2
<i>total reductions</i>	9	5

Number of combinator reductions for Ex0.

<i>combinator objects</i>	<i>SKI-combinator implementation</i>	<i>Lambda-lifting implementation</i>
<i>Plus</i>	1	1
<i>Minus</i>	1	1
<i>Constant</i>	3	3
<i>B</i>	1	0
<i>C</i>	2	0
<i>Cprime</i>	1	0
<i>Application</i>	17	7
<i>Ex1sc1</i>	0	1
<i>Ex1sc2</i>	0	1
<i>total reductions</i>	26	14

Number of combinator reductions for Ex1. Ex1sc1 & Ex1sc2 denotes supercombinators. S, B, C, Sprime, Cprime denotes standard combinators.

<i>combinator objects</i>	<i>SKI-combinator implementation</i>	<i>Lambda-lifting implementation</i>
<i>Times</i>	3	3
<i>Minus</i>	9	9
<i>Cond</i>	4	4
<i>Equal</i>	4	4
<i>Constant</i>	21	21
<i>S</i>	7	0
<i>B</i>	7	0
<i>C</i>	17	0
<i>Application</i>	137	48
<i>Dummy</i>	3	0
<i>Factscl</i>	0	4
<i>total reductions</i>	212	93

Number of combinator reductions for Fact 3. Factscl is the only super-combinator.

<i>combinator objects</i>	<i>SKI-combinator implementation</i>	<i>Lambda-lifting implementation</i>
<i>Plus</i>	2	2
<i>Minus</i>	3	3
<i>Cond</i>	6	6
<i>equal</i>	6	6
<i>Constant</i>	18	18
<i>S</i>	6	0
<i>B</i>	9	0
<i>C</i>	11	0
<i>Sprime</i>	7	0
<i>Cprime</i>	8	0
<i>Application</i>	178	49
<i>Dummy</i>	3	0
<i>Acksc1</i>	0	4
<i>Acksc2</i>	0	1
<i>total reductions</i>	257	89

Number of combinator reductions for Ack 1 1. Acksc1 & Acksc2 denotes supercombinators.

D1.4: Class representation over Instance representation of combinator objects

The result of benchmarks on Rekursiv and RISC are presented. From these tables we may note that, the number of reductions per 100 sec run with Class representation of combinator objects is improved since less time is wasted on the creation of unnecessary objects.

<i>Rekursiv</i> <i>Ex0</i>	<u><i>Instance representation</i></u>			<u><i>Class representation</i></u>		
	runs	reductions	O.created	runs	reductions	O.created
<i>SKI</i>	107560	968040 (9)	968040 (9)	124040	1116360 (9)	868280 (7)
<i>λ-lift</i>	203371	1016855 (5)	1016855 (5)	220659	1103295 (5)	882636 (4)

No. of objects created and No. of reductions per 100 sec run for Ex0 on the Rekursiv. ‘()’ denotes No. of reduction or No. of objects creation per run.

<i>RISC</i> <i>Ex0</i>	<u><i>Instance representation</i></u>			<u><i>Class representation</i></u>		
	runs	reductions	O.created	runs	reductions	O.created
<i>SKI</i>	16059	144531 (9)	144531 (9)	17747	159723 (9)	124249 (7)
<i>λ-lift</i>	31640	158200 (5)	158200 (5)	35154	175770 (5)	140616 (4)

No. of objects created and No. of reductions per 100 sec run for Ex0 on the RISC. ‘()’ denotes No. of objects creation/reductions per run.

<i>Rekursiv</i>	<u>Instance representation</u>			<u>Class representation</u>		
<i>Ex1</i>	runs	reductions	O.created	runs	reductions	O.created
<i>SKI</i>	32900	855400 (26)	855400 (26)	39949	1038674 (26)	798980 (20)
<i>λ-lift</i>	70128	981792 (14)	981792 (14)	84827	1187578 (14)	848270 (10)

No. of objects created and No. of reductions per 100sec run for Ex1 on the Rekursiv. ‘() denotes *No. of reduction or No. of objects creation per run.*

<i>RISC</i>	<u>Instance representation</u>			<u>Class representation</u>		
<i>Ex1</i>	runs	reductions	O.created	runs	reductions	O.created
<i>SKI</i>	5214	135564 (26)	135564 (26)	5761	149786 (26)	115220 (20)
<i>λ-lift</i>	10594	148316 (14)	148316 (14)	12213	170982 (14)	122130 (10)

No. of objects created and No. of reductions per 100sec run for Ex1 on the RISC. ‘() denotes *No. of reduction or No. of objects creation per run.*

<i>Rekursiv</i>	<u>Instance representation</u>			<u>Class representation</u>		
<i>Fact 3</i>	runs	reductions	O.created	runs	reductions	O.created
<i>SKI</i>	5232	1109184 (212)	523200 (100)	6239	1322668 (212)	555271 (89)
λ -lift	9876	918468 (93)	7980080 (80)	13294	1236342 (93)	717876 (54)

No. of objects created and No. of reductions per 100sec run for Fact 3 on the Rekursiv. '()' denotes No. of reduction or No. of objects creation per run.

<i>RISC</i>	<u>Instance representation</u>			<u>Class representation</u>		
<i>Fact 3</i>	runs	reductions	O.created	runs	reductions	O.created
<i>SKI</i>	940	199280 (212)	94000 (100)	983	208396 (212)	87487 (89)
λ -lift	1755	163215 (93)	140400 (80)	2013	187209 (93)	108702 (54)

No. of objects created and No. of reductions per 100sec run for Fact 3 on the RISC. '()' denotes No. of reduction or No. of objects creation per run.

<i>Rekursiv</i> <i>Ack 1 1</i>	<u><i>Instance representation</i></u>			<u><i>Class representation</i></u>		
	runs	reductions	O.created	runs	reductions	O.created
<i>SKI</i>	2776	713432 (257)	519112 (187)	3349	860693 (257)	532491 (159)
λ -lift	5999	533911 (89)	989835 (165)	7821	696069 (89)	977625 (125)

No. of objects created and No. of reductions per 100sec run for Ack 1 1 on the Rekursiv. ‘() denotes *No. of reduction or No. of objects creation per run.*

<i>RISC</i> <i>Ack 1 1</i>	<u><i>Instance representation</i></u>			<u><i>Class representation</i></u>		
	runs	reductions	O.created	runs	reductions	O.created
<i>SKI</i>	611	157027 (257)	114257 (187)	649	166793 (257)	103191 (159)
λ -lift	962	85618 (89)	158730 (165)	1144	101816 (89)	143000 (125)

No. of objects created and No. of reductions per 100sec run for Ack 1 1 on the RISC. ‘() denotes *No. of reduction or No. of objects creation per run.*

D1.5: The garbage collection overhead on the RISC.

<i>Observation Number (see appendix D1.7)</i>	<i>Executions per 100 sec</i>	<i>Execution time per run (μ-sec) measured</i>	<i>Execution time per run (μ-sec) calculated</i>	<i>Overhead factor</i>
33	16059	6227	5749	0.08
34	17747	5636	5189	0.08
35	31640	3161	2895	0.08
36	35154	2845	2615	0.08
37	5214	19179	17589	0.08
38	5761	17358	15909	0.08
39	10594	9439	8706	0.08
40	12213	8188	7586	0.07
41	940	106400	97520	0.08
42	983	101735	94440	0.07
43	1755	56975	52769	0.07
44	2013	49683	45489	0.08
45	611	163660	150276	0.08
46	649	154023	142436	0.08
47	962	103900	91584	0.12
48	1144	87447	80384	0.08

D1.6: The overhead on Rekursiv when no garbage collection is required.

<i>Observation Number (see appendix D1.7)</i>	<i>Number of executions</i>	<i>Time sec</i>	<i>Execution time per run (μ-sec) measured</i>	<i>Execution time per run (μ-sec) calculated</i>	<i>Overhead factor</i>
1	3414	1.912	560	555	0.01
2	3819	1.901	498	492	0.01
3	6112	1.842	301	300	0.00
4	6570	1.830	278	270	0.02
5	1782	2.954	1658	1630	0.01
6	1335	1.965	1472	1442	0.02
7	2296	1.940	845	844	0.00
8	2680	1.931	720	718	0.00
9	421	3.989	9475	9328	0.01
10	218	1.994	9148	8982	0.01
11	377	1.990	5279	5219	0.01
12	449	1.988	4428	4401	0.01
13	212	2.995	14125	13791	0.02
14	151	1.996	13219	12910	0.02
15	241	1.994	8273	8232	0.01
16	285	1.993	6992	6974	0.00

D1.7: A table of 48 observations for the analysis of variance for multi-way (architectures, implementation techniques, object-oriented styles, benchmarks) with their interaction factors

<u>Obser- vations</u> (Row)	<u>Time</u> per run μ -sec	<u>Style</u> 1= <i>Instance</i> 2= <i>Class</i>	<u>Technique</u> 1= <i>SKI</i> 2= λ	<u>Benchmark</u> 1= <i>Ex0</i> 2= <i>Ex1</i> 3= <i>Fact 3</i> 4= <i>Ack 1 1</i>	<u>Machines</u> 1= <i>Rek no gc</i> 2= <i>Rek with gc</i> 3= <i>RISC</i>
1	553	1	1	1	1
2	490	2	1	1	1
3	299	1	2	1	1
4	267	2	2	1	1
5	1626	1	1	2	1
6	1437	2	1	2	1
7	840	1	2	2	1
8	714	2	2	2	1
9	9277	1	1	3	1
10	8931	2	1	3	1
11	5168	1	2	3	1
12	4350	2	2	3	1
13	13768	1	1	4	1
14	12887	2	1	4	1
15	8209	1	2	4	1
16	6951	2	2	4	1
17	930	1	1	1	2
18	806	2	1	1	2
19	492	1	2	1	2
20	453	2	2	1	2
21	3039	1	1	2	2
22	2503	2	1	2	2

<u>Observations</u> (Row)	<u>Time</u> per run μ -sec	<u>Style</u> 1=Instance 2=Class	<u>Technique</u> 1=SKI 2= λ	<u>Benchmark</u> 1=Ex0 2=Ex1 3=Fact 3 4=Ack 1 1	<u>Machines</u> 1=Rek no gc 2=Rek with gc 3=RISC
23	1426	1	2	2	2
24	1179	2	2	2	2
25	19113	1	1	3	2
26	16028	2	1	3	2
27	10125	1	2	3	2
28	7522	2	2	3	2
29	36023	1	1	4	2
30	29861	2	1	4	2
31	16669	1	2	4	2
32	12786	2	2	4	2
33	6227	1	1	1	3
34	5635	2	1	1	3
35	3161	1	2	1	3
36	2845	2	2	1	3
37	19179	1	1	2	3
38	17358	2	1	2	3
39	9439	1	2	2	3
40	8188	2	2	2	3
41	106400	1	1	3	3
42	101735	2	1	3	3
43	56975	1	2	3	3
44	49683	2	2	3	3
45	163660	1	1	4	3
46	154023	2	1	4	3
47	103900	1	2	4	3
48	87447	2	2	4	3

D1.8: Analysis of Variance

The model (anova) deals with three order interactions because it involves with four main effects for example: architectures, techniques, styles and benchmarks. The expected means, estimated variance components, plots and the residual, fitted values for both main effects and their interactions are listed below in a sequential order.

Factors in model

Our model is designed with four factors.

1. Four benchmark functions

- 1 = *Ex0*
- 2 = *Ex1*
- 3 = *Fact 3*
- 4 = *Ack 1 1*

2. Three machines

- 1 = Rekursiv disregarding garbage collection and object squeeze
- 2 = Rekursiv with garbage collection and object squeeze
- 3 = RISC

3. Two implementation techniques

- 1 = SKI-combinator
- 2 = λ -lifting

4. Two object-oriented styles

- 1 = Instance representation
- 2 = Class representation

A total of 48 observations ($2 * 2 * 3 * 4 = 48$) has been recorded for this model. All timing measurements are in μ -sec. The table of 48 observations results are given on appendixD1.6 and their interaction factors are already given on previous pages.

All the interactions are indicated with asterisks (*) in the following anova tables. For example, '*instclas*skilam*' is the interaction between factors *instclas* and *skilam*.

<p style="text-align: center;">Anova Table: 1 Analysis of Variance for time (<i>Main Anova Table</i>)</p>					
<i>Source</i>	<i>DF</i>	<i>SS</i>	<i>MS</i>	<i>F</i>	<i>P</i>
a. <i>instclas</i>	1	81169408	81169408	62.59	0.000
b. <i>skilam</i>	1	2301883904	2301883904	1774.98	0.000
c. <i>benchmk</i>	3	2.1603E+10	7201128448	5552.77	0.000
d. <i>machine</i>	2	2.5468E+10	1.2734E+10	9819.34	0.000
e. <i>instclas*skilam</i>	1	805231	805231	0.62	0.461n.s
f. <i>instclas*benchmk</i>	3	71949424	23983142	18.49	0.002
g. <i>instclas*machine</i>	2	47470852	23735426	18.30	0.003
h. <i>skilam*benchmk</i>	3	1636460544	545486848	420.62	0.000
i. <i>skilam*machine</i>	2	1923825024	961912512	741.73	0.000
j. <i>benchmk*machine</i>	6	2.0286E+10	3381064448	2607.13	0.000
k. <i>instclas*skilam*benchmk</i>	3	1861422	620474	0.48	0.709n.s
l. <i>instclas*skilam*machine</i>	2	4463935	2231968	1.72	0.257n.s
m. <i>instclas*benchmk*machine</i>	6	42873588	7145598	5.51	0.028
n. <i>skilam*benchmk*machine</i>	6	1263522304	210587056	162.38	0.000
<i>Error</i>	6	7781116	1296853		
<i>Total</i>	47	7.4742E+10			

The terms used on above columns are described as:

Source : description of interactions of various factors

DF : degrees of freedom, it is one less than the number of effects e.g. there are 4 benchmark effects so DF is 3.

$$\text{Total } SS = \text{sum of } (\text{observation} - \text{mean of all observations})^2$$

e.g. Total SS = (First obs. - mean of 48 obs.)² +(second obs.....

SS : sum of squares is partitioned by the analysis of variance.

MS : mean square (SS/DF)

F : Fisher's F statistic,

$$\text{Fisher's } F \text{ for an effect} = \frac{\text{the } MS \text{ for the effect}}{\text{the } MS \text{ for error}}$$

Error : error is the estimate of variance not explained by any of the effects in the table.

P : probability that *F* would be as large as this if the effect were zero. If $P < 0.05$ then we say the effect is significant at 5%.

n.s : indicates non significant effect

The main anova table:1 shows

- four main effects a, b, c, d
- six 2-factor interactions e, f, g, h, i, j
- four 3-factor interactions k, l, m, n

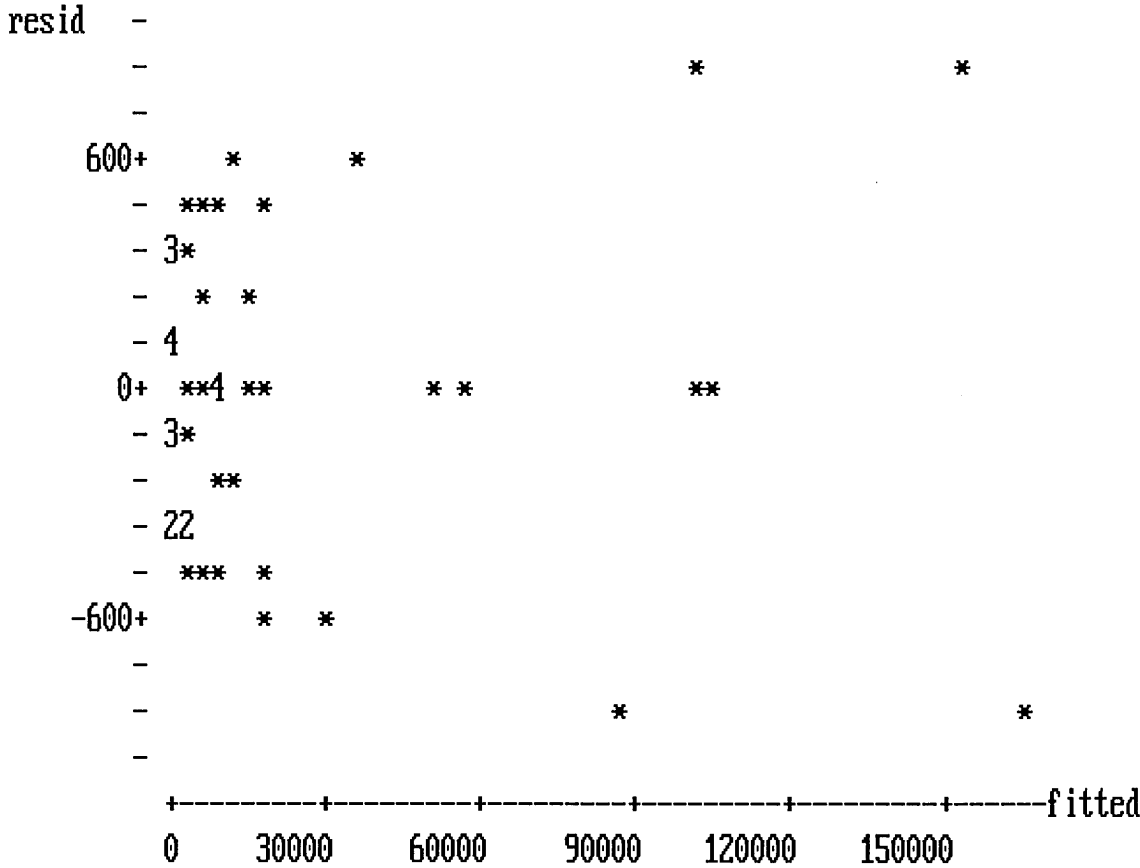
As can be seen,

- all main effects are significant.
- All but one of the 2-factor interactions are also significant.
- Two of the 3-factor interactions are also significant.

We can use a table of means to get an estimate of what the time should be for any combination of factor levels according to the model which allows for interactions upto third order (eg machine 1, technique SKI, style Instance, benchmark 1). We also know what the time was that was observed.

$$Residual\ (resid) = Observed - the\ one\ estimated\ by\ model\ (fitted)$$

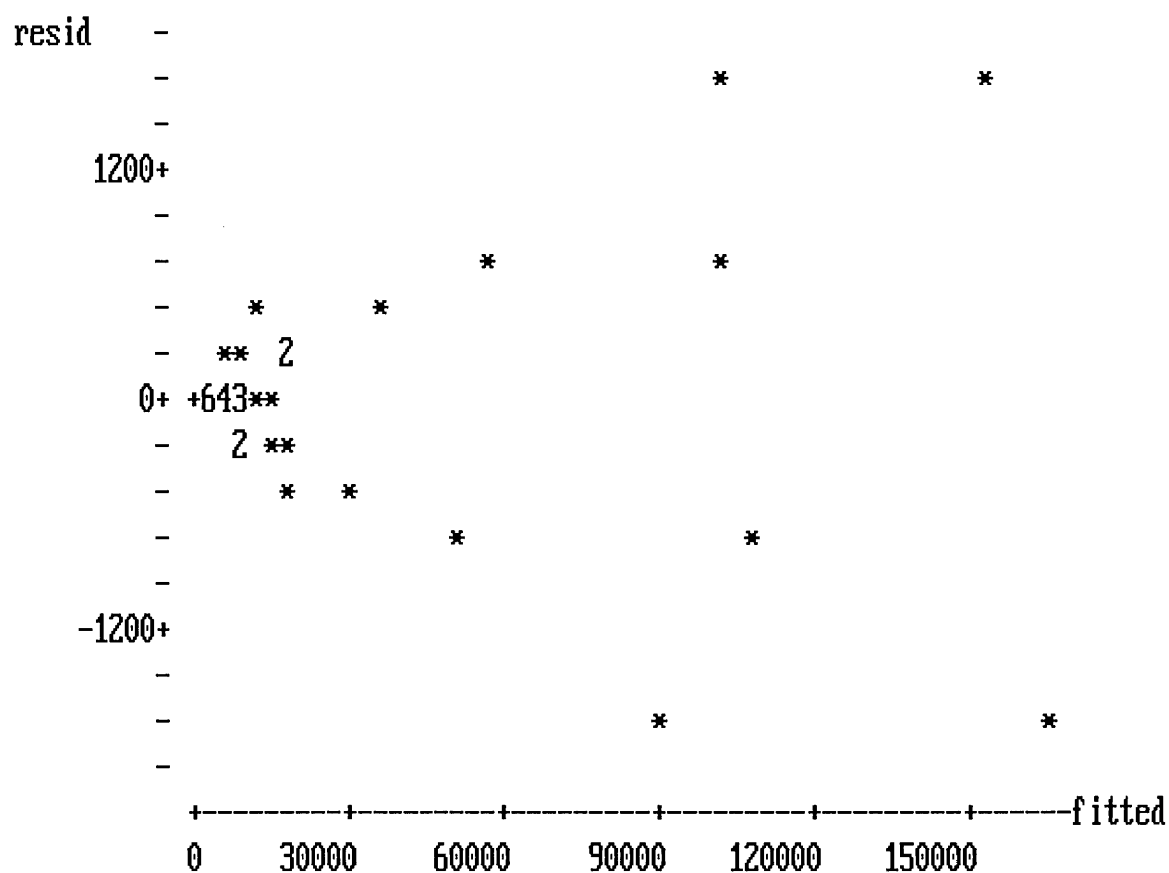
Ideally the plot of resid Vs fitted looks like a shapeless cloud since if the model is a good fit remaining variation is random. The following residual plot, 'Residual Plot: 1' is expanding as we go along for higher fitted values.



Residual plot: 1

We should try removing the insignificant effects from our anova model (Anova table:1). Removing the insignificant effects f, l, m from our main anova table:1 results the following anova table:2 and the Residual Plot:2.

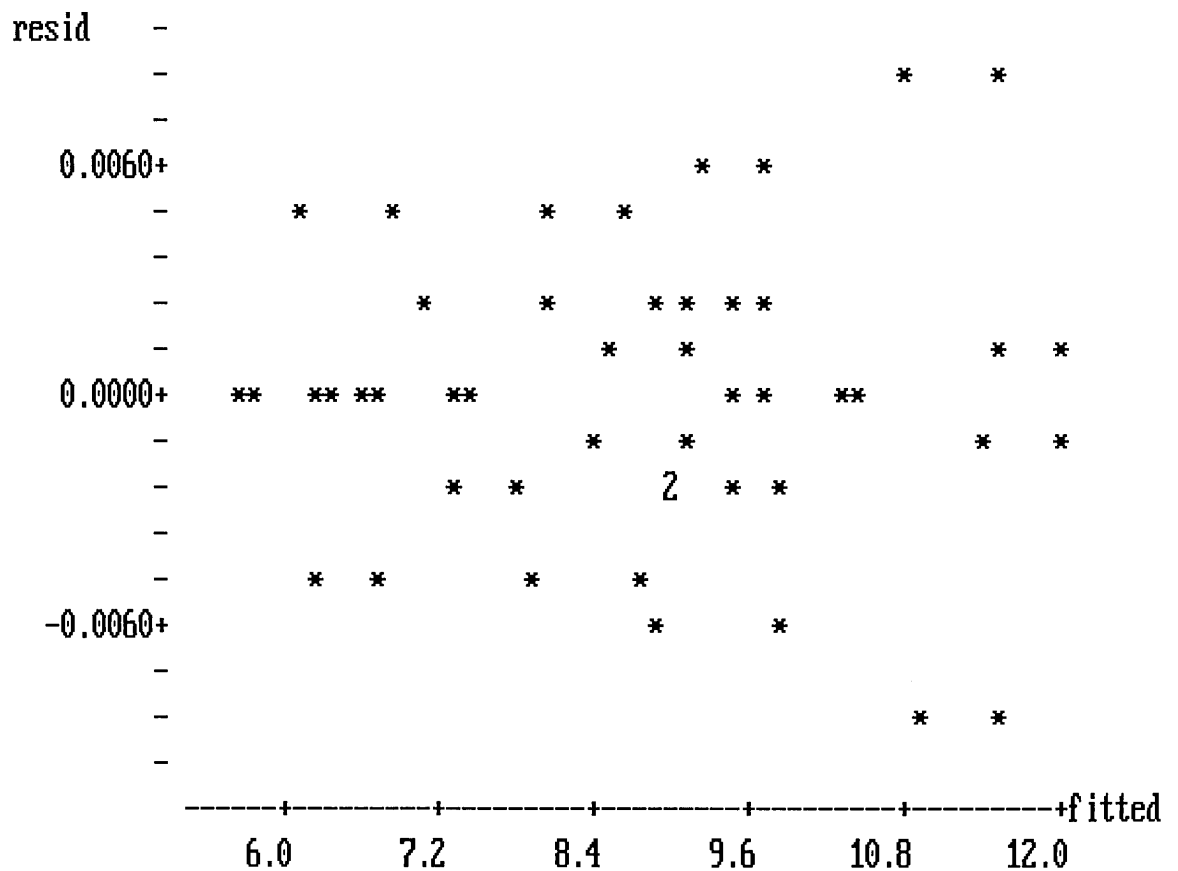
<p style="text-align: center;"><u>Anova Table: 2</u></p> <p style="text-align: center;">Analysis of variance for time (insignificant effects removed)</p>					
<i>Source</i>	<i>DF</i>	<i>SS</i>	<i>MS</i>	<i>F</i>	<i>P</i>
<i>instclas</i>	1	81169408	81169408	65.32	0.000
<i>skilam</i>	1	2301883904	2301883904	1852.41	0.000
<i>benchmk</i>	3	2.1603E+10	7201128448	5795.01	0.000
<i>machine</i>	2	2.5468E+10	1.2734E+10	1.0E+04	0.000
<i>instclas*benchmk</i>	3	71949424	23983142	19.30	0.000
<i>instclas*machine</i>	2	47470852	23735426	19.10	0.000
<i>skilam*benchmk</i>	3	1636460544	545486848	438.97	0.000
<i>skilam*machine</i>	2	1923825024	961912512	774.09	0.000
<i>benchmk*machine</i>	6	2.0286E+10	3381064448	2720.87	0.000
<i>instclas*benchmk*machine</i>	6	42873588	7145598	5.75	0.005
<i>skilam*benchmk*machine</i>	6	1263522304	210587056	169.47	0.000
<i>Error</i>	12	14911704	1242642		
<i>Total</i>	47	7.4742E+10			



Residual Plot: 2

The 'Residual Plot: 2' now clearly shows that variance increases from left to right. This kind of problem can sometimes be solved by using logs especially if the observations cover several orders of magnitude as these do.

<p style="text-align: center;"><u>Anova Table: 3</u></p> <p style="text-align: center;">Analysis of variance for logtime</p>					
<i>Source</i>	<i>DF</i>	<i>SS</i>	<i>MS</i>	<i>F</i>	<i>P</i>
<i>instclas</i>	1	0.2353	0.2353	2094.27	0.000
<i>skilam</i>	1	5.2917	5.2917	4.7E+04	0.000
<i>benchmk</i>	3	89.7335	29.9112	2.7E+05	0.000
<i>machine</i>	2	51.7687	25.8843	2.3E+05	0.000
<i>instclas*skilam</i>	1	0.0089	0.0089	78.76	0.000
<i>instclas*benchmk</i>	3	0.0036	0.0012	10.53	0.008
<i>instclas*machine</i>	2	0.0165	0.0083	73.47	0.000
<i>skilam*benchmk</i>	3	0.0168	0.0056	49.89	0.000
<i>skilam*machine</i>	2	0.0174	0.0087	77.47	0.000
<i>benchmk*machine</i>	6	0.0917	0.0153	136.00	0.000
<i>instclas*skilam*benchmk</i>	3	0.0090	0.0030	26.79	0.001
<i>instclas*skilam*machine</i>	2	0.0006	0.0003	2.89	0.132 n.s.
<i>instclas*benchmk*machine</i>	6	0.0072	0.0012	10.71	0.005
<i>skilam*benchmk*machine</i>	6	0.0404	0.0067	59.91	0.000
<i>Error</i>	6	0.0007	0.0001		
<i>Total</i>	47	147.2421			

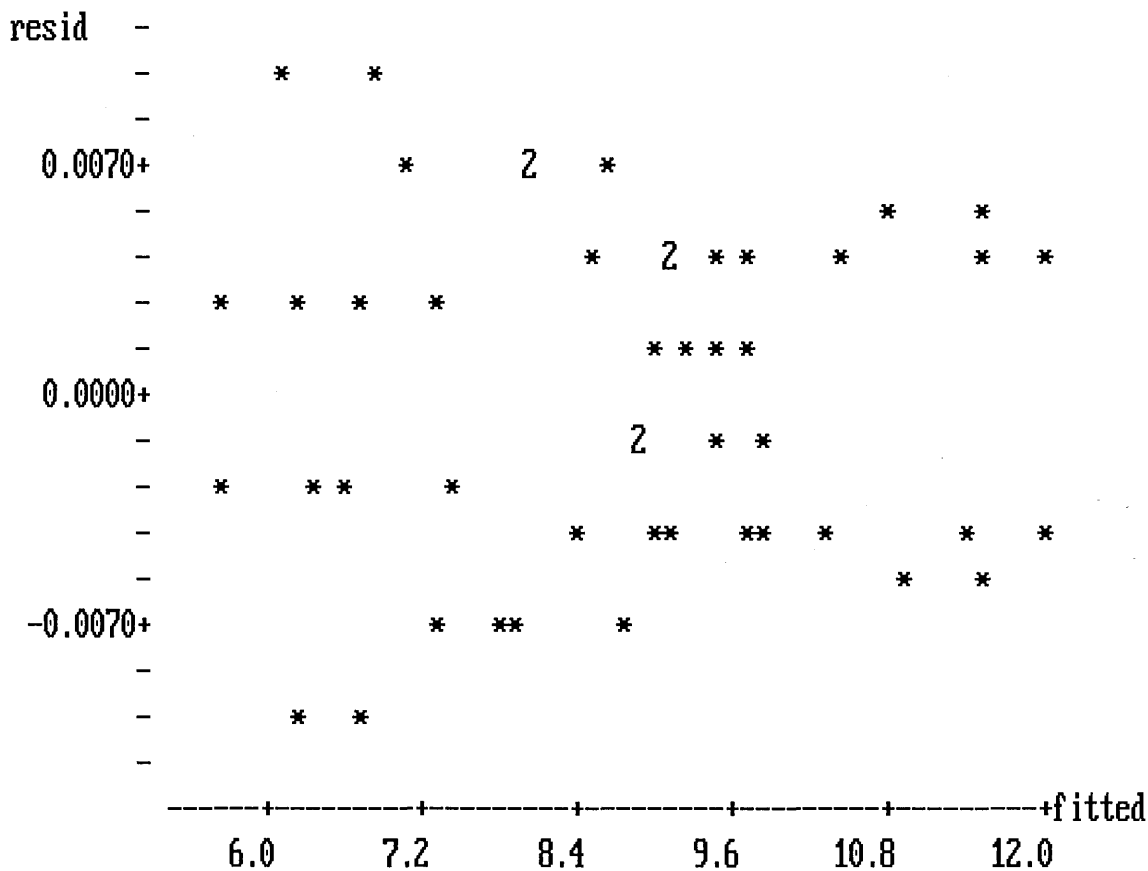


Residual Plot: 3

In this case all second order and all but one of the third order interactions are significant. The 'Residual Plot: 3' looks better than the previous one though not completely satisfactory.

Try removing the insignificant effect from anova table:3. The third order insignificant effect which is removed is (*instclas*skilam*machine*).

<p style="text-align: center;">Anova Table: 4</p> <p style="text-align: center;">Analysis of variance for time (3rd order insignificant effects removed)</p>					
<i>Source</i>	<i>DF</i>	<i>SS</i>	<i>MS</i>	<i>F</i>	<i>P</i>
<i>instclas</i>	1	0.2353	0.2353	1421.85	0.000
<i>skilam</i>	1	5.2917	5.2917	3.2E+04	0.000
<i>benchmk</i>	3	89.7335	29.9112	1.8E+05	0.000
<i>machine</i>	2	51.7687	25.8843	1.6E+05	0.000
<i>instclas*skilam</i>	1	0.0089	0.0089	53.47	0.000
<i>instclas*benchmk</i>	3	0.0036	0.0012	7.15	0.012
<i>instclas*machine</i>	2	0.0165	0.0083	49.88	0.000
<i>skilam*benchmk</i>	3	0.0168	0.0056	33.87	0.000
<i>skilam*machine</i>	2	0.0174	0.0087	52.59	0.000
<i>benchmk*machine</i>	6	0.0917	0.0153	92.34	0.000
<i>instclas*skilam*benchmk</i>	3	0.0090	0.0030	18.19	0.001
<i>instclas*benchmk*machine</i>	6	0.0072	0.0012	7.27	0.007
<i>skilam*benchmk*machine</i>	6	0.0404	0.0067	40.68	0.000
<i>Error</i>	8	0.0013	0.0002		
<i>Total</i>	47	147.2421			



Residual Plot: 4

The Residual Plot: 4 shows a tendency for variance to decrease from left to right. The four biggest residuals belong to the observations at benchmark 1 on machine 2. These four observations are not very well explained by the model.

We removed out the *instclass*machine*skilam* interaction because it was insignificant. However, it seems possible that even though this interaction is not significant there is some complex relationship between these factors which our new model fails to deal with.