A PARALLEL IMPLEMENTATION OF SASL

Jiannis Corovessis

A Thesis Submitted for the Degree of PhD
at the
University of St Andrews

1983

Full metadata for this item is available in
St Andrews Research Repository
at:
http://research-repository.st-andrews.ac.uk/

Please use this identifier to cite or link to this item:
http://hdl.handle.net/10023/13448

This item is protected by original copyright
A PARALLEL IMPLEMENTATION OF SASL

by

Jiannis Corovessis

A thesis submitted for the degree of Doctor of Philosophy

Department of Computational Science

University of St.Andrews

St.Andrews

September 1982
Declaration

I declare that this thesis has been composed by myself and that the work that it describes has been done by myself. The work has not been submitted in any previous application for a higher degree. The research has been performed since my admission as a research student under Ordinance General No 12 on 1st. October 1978 for the degree of Doctor of Philosophy.

/  

Jiannis Corovessis
I hereby declare that the conditions of the Ordinance and Regulations for the degree of Doctor of Philosophy (Ph.d.) at the University of St. Andrews have been fulfilled by the candidate, Jiannis Corovessis

A. T. Davie
ABSTRACT

The applicative or functional language, SASL, is investigated from the point of view of an implementation. The aim is to determine and experiment with a run-time environment (SASL parallel machine) which incorporates parallelism so that constituent parts of a program (its sub-expressions) can be processed concurrently.

The introduction of parallelism is characterised by two fundamental issues. The type of programs, referred to as parallel, and the so-called strategy of parallelism, employed by the parallel machine. The former concerns deriving a graph from the program text indicating the order in which things must be done and the notion of "worthwhile" parallelism. In order to obtain a parallel program, the original (sequential) program is transformed and/or modified. Certain programs are found to be essentially sequential. Parallelism is expressed as call-by-parallel parameter passing mechanism and by a parallel conditional operator, suggesting speculative parallelism.

The issue of the strategy of parallelism concerns the scheme under which a regime of SASL processors combine their effort in processing a parallel program. The objective being to shorten the length of computation carried out by the sequential machine on the initial program.

The class of parallel programs seems to be non-trivial and it includes both non-numerical and numerical programs. The "speed-up" by appealing to parallelism for such programs is found to be substantial.
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ONE</td>
<td>1</td>
</tr>
<tr>
<td>TWO</td>
<td>7</td>
</tr>
<tr>
<td>THREE</td>
<td>29</td>
</tr>
<tr>
<td>FOUR</td>
<td>55</td>
</tr>
<tr>
<td>FIVE</td>
<td>65</td>
</tr>
<tr>
<td>SIX</td>
<td>104</td>
</tr>
<tr>
<td>SEVEN</td>
<td>115</td>
</tr>
</tbody>
</table>

Appendix I

Simulation tables

Appendix II

Source program of the simulator
CHAPTER ONE

Introduction

The essence of program notations referred to as applicative or functional [1] lies in the fact that they possess the familiar properties of the notation of mathematics void of imperative notions. This is the point of difference with conventional program notations referred to as imperative, examples of which are the languages FORTRAN, ALGOL, PASCAL etc, where the basic programming notions are sequencing and store manipulation.

The meaning of a program in an imperative language is the behaviour (history of states) traced by the underlying mechanism executing the program (given its data). Certain states involve performing input or output. Thus computation is expressed in terms of state changes. Each constituent part of the program has to wait for the appropriate state of the machine to be arrived at before it makes its contribution.

The execution of a part may cause a "side-effect" on the execution of another. The presence of side-effects causes the underlying mechanism to be sequential, performing one thing at a time.

On the contrary the meaning of an applicative program is an object in the universe of discourse of the language. The object is referred to as the value of the program. At this level of programming the behaviour of the underlying
mechanism evaluating the program is not addressed by the program.

The computation that an applicative program entails is a transformational process [2] of the program (data is part of the program in applicative languages) through a sequence of intermediate representations of its value to a final representation, providing the sequence converges. This is a canonical representation of the value the program denotes. The value is sometimes referred to as its Normal Form [3]. Obtaining this representation achieves termination of the computation since no further transformation is possible.

Computation as a transformation process suggests that there is an invariance relation between the states the evaluation mechanism traces, namely that the meaning of the program is preserved at all times. What actually changes between states (otherwise the machine would be of no use) is the representation of the program's value. Each transformation results in more detail about the canonical representation being computed. Note we refer to evaluation of an applicative program and to execution of an imperative program for obvious reasons. An imperative program's result is obtained as a side-effect during its execution.

The contribution of a constituent part of a program is also a value which results after it has been transformed (simplified) to the canonical representation of this value. Thus the evaluation of this part has no "effect" on the
evaluation of another part. Its purpose is to communicate this value.

We observe the evaluations a program entails are partially ordered with respect to the data dependencies between them. An evaluation is data dependent on another when the latter is a sub-evaluation of the former. This can be determined from the program text, represented as a graph, as will be shown in chapter five.

The standard evaluation mechanisms of applicative languages flatten this partial order to a total order bound by the uni-processor implementation environment. The objective of the present study is the construction of an evaluation mechanism which exploits the "inherent" parallelism of programs, suggested by the partial order and the type of parallelism possessed by various programs.

The investigation is based on the applicative language SASL [4]. The work is organised into chapters as follows.

Chapter two consists of three sections. The first introduces the basic features of SASL, the second describes its computational process and the mathematical properties it possesses and the third introduces the notion of parallel operators and a call-by-parallel parameter passing mechanism. Annotations in the program text are used to specify parallel primitive operators and a system function STRICT encapsulates the parallel parameter passing mechanism. Both parallel primitive operators and the (meta-)
function \textsc{strict} can be thought of as "hints" to the evaluation mechanism, specifying possible parallel behaviour.

In chapter three the parallel implementation of SASL developed in S-ALGOL [5] is described. This is based on an earlier implementation of SASL [6]. The evaluator of this SASL system was unsuitable for our purpose so a new parallel evaluation mechanism was constructed and interfaced to the rest of the SASL system. This gave us a complete SASL system to develop and experiment with parallel programs. The level of parallelism concerns the concurrent progress of evaluators each carrying out part of the computation a program entails.

Chapter four describes the model of parallelism where the evaluation mechanism employs different strategies for parallelism. A strategy of spawning, as it is referred to, determines when an assistant evaluator is to be assigned a task. Initially there is a single task and one evaluator. The presence of parallel operators in the program text, translated by the compiler into parallel instructions generate a tree of tasks, which represents the partial order mentioned previously.

A strategy causes either the realisation of the partial order by employing an evaluator on each branch of the tree or its conversion to a total order where a single evaluator traverses the tree simulating the parallel evaluation of
tasks. No bound on the number of evaluators is assumed.

The effect of parallelism is measured in terms of the evaluation steps [7] a program entails. This characterises the performance of programs so that comparison between different strategies can be made. The results of the parallel evaluation of programs are presented in graphical form in the appendix.

In chapter five the idea of parallelisation of programs is put forward whereby a SASL program written initially without consideration of the evaluation mechanism is transformed into a parallel program. Work has been done by Darlington and Burstall [8] on automatic and semi-automatic transformations. None of that is implemented here. Transformations to suitable forms is by hand directed by program graphs (see next paragraph). A parallel program as opposed to a sequential program is one whose evaluation splits into a number of sub-evaluations, each of which may decompose further. Programs structured in this way result from adopting a programming style known as Divide-and-Conquer [9].

In order to identify the sub-evaluations which can be carried out in parallel a program text is represented as a directed graph. The nodes of the graph correspond to operators in the language which construct composite parts (expressions). The arcs of the graph show the data dependencies between evaluations of parts. Arcs out of a
node which do not converge onto a common node characterise the corresponding operator as parallel. This implies its operands may be computed in parallel.

A number of graph manipulation, list processing, numerical and symbol manipulation parallel programs have been developed. The parallelism of these programs is investigated by submitting them to the evaluation mechanism under different strategies of spawning.

Chapter six contains the results of testing the example programs developed in chapter five on the parallel evaluator described in chapter three, under different strategies as described in chapter four.

Conclusions are presented in chapter seven.
CHAPTER TWO

The language SASL, its computational process and its extension by parallel constructs

In this chapter we present the basic features of the applicative/functional language SASL (more details of which can be found in [4]), the computational process it entails and its extension with constructs which express a certain notion of parallelism. The name SASL stands for "St. Andrews Static Language". "Static" refers to the fact that SASL contains no commands and a data structure once created cannot be altered. "Applicative" or "functional" indicates the programming style of the language where algorithms are specified in terms of functions applied to arguments as the only "control" construct available to the programmer.

A SASL program is an expression which has for its value an object. The outcome of the program is to print a representation of the object, unlike the programs in imperative languages like FORTRAN, ALGOL etc. where a program specifies what behaviour step-by-step the machine is to perform, each construct in the program addresses the "state" of the machine.

In SASL algorithms are specified at an abstraction level over the state of the machine which executes the program. In fact all the states the machine traces in executing (evaluating) a program from the initial one to the final one are equivalent from the point of view that each preserves
the meaning (value) of the program. Thus the state of the machine is not addressed in the program. The programmer computes with objects rather than with states. Note that although all machine states preserve the meaning of the program (and data) they are actually different states since each new state must be associated with computing some detail about the program's value not present in the previous state, otherwise the machine would be of no use.

Objects

The data items SASL expressions describe are called objects. Every expression has an object as its value. No significance is attached to expressions other than as a means to talk about objects. Any sub-expression expression can be replaced by any other which has the same value without affecting the value of a larger expression of which is is a part. This is a property of expressions called Referential Transparency [10].

The universe of discourse of SASL has six types of object:

1. Numbers - these are the integers such as -5, 0, 99 etc.

2. Truth-values - there are two such objects true and false.

3. Characters - %a represents the character a, % represents the character % etc.
4. Lists - a list is an ordered collection of objects called its components. For example

1, 2, 1 and 99,

are lists of length 3 and 1 respectively. Note that repetition of components is allowed. A list may have an infinite number of components. For example the list of all integers is a well defined SASL object. The empty list is represented by the constant ().

5. Functions - a function is a rule which associates to a SASL object (the input of the function) a unique SASL object (its output).

6. Undefined - there is a unique object undefined which is the value of expressions such as %a+1 and of expressions which entail non-terminating computations. Note here we differentiate between a non-terminating computation and a computation whose result is an infinite list. An infinite list is a perfectly well defined object but only a finite number of its components can be printed in finite time.

The language obeys the rule that all six types of object have the same "civil rights" :-

any object can be named
any object can be a component of a list
any object can be the input of a function
any object can be the output of a function
the above rule characterises the language as being semantically complete [11].

Expressions

Expressions are either atomic (they have no syntactic structure for example a constant or a name) or they are composite (constructed out of sub-expressions. The usual arithmetic, logical and relational operators construct one sort of composite expressions.

Juxtaposition of two expressions, for example

$$A \ B$$

denotes the application of a function to its argument (the input of the function). It also denotes selection from a list. For example

$$(1, 99, 4) \ 2$$

selects 99, the second component of the list.

List expressions are constructed with the operator $:\$, for example

$$x : \text{list}$$

constructs a new list by prefixing to list the component $x$. 
Commas are shorthand for list expressions. For example

\[ 1, 2, 3 \text{ and } 1:2:3:() \]

are equivalent. Concatenation of two lists is denoted by the operator ++. For example

\[ (1,2) ++ (3,) \]

gives the list 1, 2, 3

Another form of composite expression is the conditional expression constructed with the operator →. For example

\[ A \rightarrow B ; C \]

denotes the value of B or C respectively depending on whether the value of A is true or false, otherwise it denotes the object undefined.

An expression may include definitions of names that appear in it using the where construct followed by clauses. Each clause defines a name. For example

\[ a + b \]

\[ \text{where} \]

\[ a = 1 \]

\[ b = 2 \]

evaluates to 3.
Nested definitions are allowed, for example

\[
a + b
\]
\[
\text{where}
\]
\[
a = 1
\]
\[
b = 2 + c
\]
\[
\text{where}
\]
\[
c = 3
\]
evaluates to 6. Multiple definitions are also possible, for example

\[
a, b, c = 1, 2, 3
\]
is equivalent to

\[
a = 1
\]
\[
b = 2
\]
\[
c = 3
\]

Definitions in general are of the form \( \text{LHS} = \text{RHS} \) where \( \text{LHS} \) is a construction of arbitrary complexity built from names and constants using commas, brackets and the operator :. The \( \text{RHS} \) varies over expressions, for example

\[
x : y = 1, 2, 3, 1+3
\]
is equivalent to

\[
x = 1
\]
\[
y = 2, 3, 1+3
\]
the name \( y \) denotes the list \( 2, 3, 4 \).
In the case of function definition, \LISP consists of the name of the function being defined followed by one or more formal parameters. As a formal parameter we can have a name, a constant or a construction of arbitrary complexity enclosed in brackets, as in multiple definitions. Names in formal parameters denote arbitrary input objects and they are local to the clause. For example the clause

$$\text{sum } (x,y) = x+y$$

defines the function which computes the sum of two integers, passed to it as the components of a 2-list. Another way of defining the same function is

$$\text{sum } x \ y = x+y$$

where we use the fact that a function (denoted by \text{sum } x) can return as its value another function (that which adds \textit{x} to its parameter). Note that the (more general) function \text{sum} can be partially parameterised \cite{12} to yield the (less general, specialised) functions \text{incr} and \text{decr}

$$\text{incr } = \text{sum } 1$$
$$\text{decr } = \text{sum } (-1)$$

so that the expressions \text{incr } 1 and \text{decr } 1 evaluate to 2 and 0 respectively.
Functions can be defined by more than one clause each clause covering a case of the parameters, for example the clauses

\[
\text{LENGTH}() = 0 \\
\text{LENGTH}(a:x) = 1 + \text{LENGTH }x
\]

define the function which computes the length of its input list. The first clause applies to the case where the input is the empty list. In the second clause the input is a list where the name \(a\) denotes the first component of the list and \(x\) denotes the list of the rest of the components. Clauses are ordered by the order they are written. Thus in the example below

\[
\text{factorial} 0 = 1 \\
\text{factorial } n = n \times \text{factorial} (n-1)
\]

the function \(\text{factorial}\) expects its actual parameter to be the object zero or an arbitrary object, in that order. Definitions of names as well as definitions of functions can be circular too, for example

\[
\text{ones} = 1 : \text{ones}
\]

defines the infinite list \(1:1:1:...\). Definitions can also be mutually recursive, as in the following program
list1

where

list1 = 1 : list2
list2 = 2 : list1

the above program denotes the infinite list 1: 2: 1: 2...

The syntax of the language obeys the rule that any expression can be a sub-expression of a composite expression. Wrapping up an expression in brackets does not have any effect on its value, it merely affects the syntax.

Computational process

The use of = in definitions of the form LHS = RHS has two important consequences

(a) It allows an equational proof theory [13] to be built where facts we wish to prove about programs are stated as equations (clauses) in the same language as the programs are written in. The clauses are used as axioms to derive a fact which holds for a program.

(b) It characterises the mechanism of computation the language entails based on the notion of substitution where every instance of the form LHS in an expression is replaced by the RHS providing the scope of names is taken into account in the obvious way. The substitution operation plus those operations such as +, * etc. determine how a computation gets done. We shall discover that this mechanism is flexible enough to allow the introduction of
parallelism where the operations along the computation path overlap in time by splitting the computation path into parallel sub-paths. Consider the program for the factorial function, using each clause of the definition of function factorial as a substitution rule and arithmetic rules as simplification rules the computation path the program entails is shown in figure 2.1

\[
\text{factorial } 3 \\
3 \times \text{factorial } 2 \\
3 \times 2 \times \text{factorial } 1 \\
3 \times 2 \times 1 \times \text{factorial } 0 \\
3 \times 2 \times 1 \times 1 \\
6
\]

figure 2.1 - a computation path

note each substitution produces a refinement (simplification) of the representation of the object 6. We refer to the above process as being carried out by an "evaluator" for the language.
The evaluator comes up against the problem of which substitution to perform whenever there is a choice, as in the following program

\[ g(n) \text{ factorial} (-1) \]

where

\[ g(x) = 1 \]

if the inner substitution is always preferred the path diverges as shown in figure 2.2

\[ g(n) \text{ factorial} (-1) \]
\[ g(-1 \times \text{ factorial} (-2)) \]
\[ g(-1 \times -2 \times \text{ factorial} (--5)) \]

figure 2.2 - inner substitutions, divergent path

if the outer substitution is performed the path converges to 1 in one step, figure 2.3

\[ g(\text{ factorial}(-1)) \]

1

figure 2.3 - outer substitution, convergent path

Another problem with substitutions is the possibility of different paths converging to different results. Both
of the above problems are answered in the context of formal systems such as the Lambda Calculus [3] and SRS [14]. The Lambda Calculus is the basis of SASL and other applicative languages [15,16,17]. It is a formal system where concepts such as variable binding and variable abstraction can be studied but it is not a programming language because it lacks a definite universe of discourse. The entities referred to as functions in the Lambda Calculus have general character since they do not express a relation between some definite objects. The introduction into the Lambda Calculus of objects with their associated operations, like those supported by SASL, plus "syntactic sugar" gives a programming language, namely SASL. Mathematical results which hold in the Lambda Calculus by implication are assumed to hold for SASL too, although strictly speaking it must be proved they also hold for objects and operations introduced into the Lambda Calculus. Computation in the Lambda Calculus is carried out in terms of transforming an expression to another by applying certain rules, called reduction rules. These are concerned with renaming names occurring in an expression, simplification of certain expressions and substitution of an expression for the occurrences of a name in an expression. An expression which cannot be transformed any further by application of the reduction rules is said to be in Normal Form. Computation with an expression is a sequence of reduction rules applied to the expression. A finite sequence, producing a Normal Form of the expression,
represents a terminating computation.

The central result in Lambda Calculus is the Church-Rosser theorem [18] which states that for expressions A, B, C if A reduces to B and A reduces to C then there exists an expression D to which both expressions B and C reduce. This is diagramatically represented by completing the diamond where the arrow represents the application of a reduction rule, figure 2.4

\[ \begin{array}{c}
A \\
B \\
D \\
C
\end{array} \]

figure 2.4

a corollary of the Church-Rosser theorem guarantees uniqueness of Normal Forms. If two different computation paths which an expression gives rise to terminate, they do so with the same Normal Form. The Church-Rosser theorem secures independence from the order in which evaluations are carried out, except in the cases where the (meta-) algorithm driving the evaluator imposes a particular order so as to ensure that a non-terminating path is not chosen at the expense of a terminating path.

An algorithm known as Normal Order Reduction which always performs the outermost leftmost reductions first is proved to achieve termination providing there is a Normal Form for the expression [3]. This is reflected in SASL by
adopting a parameter passing mechanism referred to as call-by-need \[19\] where actual parameter expressions are passed unevaluated (no substitutions done on them) to the function. Thus the clause

\[ f \ x = 1 \]

defines a proper object (a function) even in the case where \( x \) denotes the object undefined.

From the point of view of the proof theory this is necessary in order to use

\[ = \]

as it is used in mathematics. Formally this is stated as the equality being fully substitutive \[13\].

Consider the following program

\begin{verbatim}
factorial 4
where
factorial n = fsplit 1 n
fsplit i i = i
fsplit i j = split i mid *
    fsplit (mid+1) j
where
    mid = (i+j)/2
\end{verbatim}

at each occurrence of the operator \(*\) we can split the computation path into parallel paths, see figure 2.5.
factorial 4

\[
\text{fsplit 1 2 * fsplit 3 4}
\]

\[
\text{fsplit 1 1 * fsplit 2 2 \quad fsplit 3 3 * fsplit 4 4}
\]

\[
1 \quad 2 \quad 3 \quad 4
\]

\[
2 \quad 12
\]

figure 2.5 - splitting a computation path

Thus the Church-Rosser theorem gives rise to the possibility of several evaluators working simultaneously, each pursuing a sub-path of the computation a program entails. This brings us to the subject of this thesis which is to devise and experiment with such a mechanism.

**Parallel operators and Call-by-parallel**

In the previous section the possibility of parallelism was noted where a computation path splits (see figure 2.5) when an operator expression of the form \( A*B \) is evaluated. In order to identify the expressions that can be evaluated in parallel a program is represented as a directed graph. A node with arcs to other nodes identifies a composite expression constructed by some operator, its arcs point to nodes which identify the operands of the operator. The Clauses (definitions) are used to unfold [8] the graph. The graph shows the structure of a program in terms of the data
dependencies between evaluations that it entails. An evaluation is data dependent on another when the former requires the result (value) of the latter. Data dependencies impose an order in which the associated evaluations must be carried out. Representing a program as a graph we see that evaluations are partially ordered with respect to the data dependencies which arise between them hence certain evaluations can be carried out in parallel. Consider the program graph, shown in figure 2.6, of the following program:

```
rec 0 = 1
rec n = x + square x

where
x = rec (n-1)

square a = a * a
```

![Program Graph of the Function "rec"](image_url)

*figure 2.6 - program graph of the function "rec"*
in the graph, shown in figure 2.6, we see that both operands of the operator + depend on the evaluation of the sub-expression

$$\text{rec } (n-1)$$

and so they cannot be usefully evaluated in parallel. The same is true of the operands of *. Thus the values of the function rec for n, n-1, ... must be evaluated sequentially. Consider also the graph, shown in figure 2.7, of the function "or" (used in Example 3, chapter five), defined as

$$\text{or } m \ n = m() \rightarrow n$$

the condition, m=(), and the left alternative, n, operands of the conditional operator \(\rightarrow\), can be evaluated in parallel whereas the right alternative, m, is data dependent on the condition and hence must be evaluated after the condition has been evaluated. The operands of the relational operator = can be evaluated in parallel but the operand () entails a rather trivial evaluation offering no opportunity for useful parallelism.
Analysis of a program in this way, to discover the data dependencies and the informal analysis of what expressions are "worth" evaluating in parallel characterises the instances of operators which are to be interpreted as being parallel. The conditional operator is said to be strict in its first operand and non-strict in the second and third. Other non-strict operators are &\& and |\|
(logical and, or respectively) which have their operands evaluated in parallel. In the case of evaluating an expression of the form A&\&B termination of one of A or B, with the result false causes the evaluation of the other to become irrelevant [20] (even if its value is undefined) Thus non-strict operators involve initiating an evaluation in anticipation that its value might be needed.

Parallelism can also be manifested as parallel evaluation of the arguments of a function. For example expressions of the form

\[ \text{SUM matrix1 matrix2 k} \]

met in a program for matrix multiplication (Example 8, chapter five) where matrix1 and matrix2 are sub-expressions which may be evaluated strictly before the whole expression is evaluated. In order to secure this form of parallelism, a call-by-value parameter passing mechanism, refered to as call-by-parallel must be adopted in this case. For this

\[ f \text{ is said to be strict when } f(\text{undefined}) = \text{undefined} \text{ holds.} \]
purpose a system function \textsc{strict} is implemented which effects call-by-parallel. It is described as follows

\textsc{strict} \( f \ x \ y = \) evaluate strictly \( x \) and \( y \) and then
Evaluate the expression \( f \ x \ y \)

now the above expression becomes

\textsc{strict} \text{sum} \ matrix1 \ matrix2 \ k

where \( \text{matrix1} \) and \( \text{matrix2} \) are evaluated in parallel. Note \( k \) is not "taken in" by the function \textsc{strict}. In fact the function \textsc{strict} can be defined in terms of the parallel operator \&\# as follows

\textsc{strict} \( f \ x \ y = x=x \&\# y=y \rightarrow f \ x \ y \)

'dummy''

where the expression \( x=x \) evaluates always to true and forces strict evaluation of \( x \).

In general the pattern of the parameters which are taken will vary, for example suppose that in the following expression

\( F \ x1 \ x2 \ x3 \ x4 \)

only \( x1 \) and \( x2 \) need be evaluated (strictly) in parallel. A function \( \text{gs1} \) can be defined in terms of \textsc{strict}

\( \text{gs1} \ F \ x1 \ x2 \ x3 \ x4 = \text{strict} \ aux \ x1 \ x3 \)

where

\( aux \ a \ b = F \ a \ x2 \ b \ x4 \)
Call-by-need is being retained in all other cases of function application where parallelism is not required. "Lazy evaluation" is another name for the call-by-need mechanism, mentioned previously, concerning parameters of functions and list constructors [21, 22].

We use STRICT in a number of similar cases where operands of functions or infix operators : and ++ need to be called by value. Consider for example the function

\[
\text{FOR } a \ b \ f = a > b \rightarrow ()
\]

\[
f a : \text{FOk} \ (a+1) \ b \ f
\]

whose result is a list. Parallelism can be effected by replacing : by a function cons and forcing simultaneous call-by-value on its parameters

\[
\text{FOR } a \ b \ f = a > b \rightarrow ()
\]

\[
\text{STRICT cons}
\]

\[(f a)\]

\[(\text{FOk} \ (a+1) \ b \ f)\]

the graph of FOk is shown in figure 2.8

```
figure 2.8 - program graph of function FOk
```
however one of the parallel computations accomplishes little.

In order to balance the evaluations of the list's components FOR is modified as follows

\[
\text{SPLITFOR} \ a \ b \ f = a = b \rightarrow f \ a, \\
\text{STKICT} \\
\text{APPEND} \\
(SPLITFOR \ a \ \text{mid} \ f) \\
(SPLITFOR \ (\text{mid+1}) \ b \ f) \\
\text{where} \\
\text{mid} = \frac{a+b}{2}
\]

and its graph is shown in figure 2.9

![Figure 2.9 - Program graph of function SPLITFOR](image)

The last two cases of parallelism suggest that in order to extract parallelism lazy evaluation has to be forced to do some work. Replacing call-by-need by call-by-value cannot be introduced safely without risking non-termination [23]. The approach to effecting parallelism adopted here is based on the parallel call-by-parallel scheme and on using
annotation symbols which mark strict (infix) operators as being parallel, the parallel + operator, for example, is written as +# which the compiler takes note of and produces a parallel PLUS instruction for the evaluation mechanism.
CHAPTER THREE

Implementation

In chapter two we saw that the computational process SASL entails is based on the notion of substitution. This process is implemented on an abstract machine. "Abstract" refers to the fact that the machine's behaviour is simulated in software. An implementation of a substitution machine in hardware is reported in [24]. Substitution machines are of two basic types, characterised by the way they support the notion of substitution.

The first type consists of the Reduction machines, where substitutions are performed literally on the machine representation of a program. Each substitution results in modifying part of the representation. Termination is reached when there are no further substitutions to perform, a canonical representation of the object the program denotes has been obtained. The machine representation of a program is either a graph or a string. In graph reduction parts of the graph are shared through pointers. Reducing a shared part is felt simultaneously by all other parts which have pointers to it. In string reduction a substitution may produce multiple copies of a part and each has to be reduced separately. In graph reduction substitutions are performed on the program graph directly using the clauses (definitions) of the program as substitution rules [25] or the program is compiled into a fixed set of constants.
called combinators. This incorporates a process of removing all the variables which appear in the program based on a technique borrowed from logic [26,43]. The operation of substitution on combinatory code is much simpler than that on program graphs where attention must be paid to conflicts of names.

The second type consists of the Interpreters or fixed program machines, where substitutions are simulated [27,28,29]. The machine representation of a program remains unmodified throughout the computation but the data mutates. The source text of the program is compiled into a code tree where each node of the tree represents an instruction of the machine. This is interpreted by the machine causing it to modify its state.

The present investigation is based on fixed program machines, known as the SECD machines [28]. The state of such machines consists of a Stack, an Environment, a Control and a Dump component. SECD machines represent the original attempts to implement applicative/functional languages, influenced by the machines of algol-like languages.

We shall describe an implementation of SASL based on the SECD type machines and then we shall modify it so that several machines can combine their effort in carrying out the computation a program entails.
The SASL machine

The SASL machine is simulated by a program written in S-ALGOL. It is based on the original SASL machine [29] which supported a weaker version of SASL without infinite lists and multiple definitional Clauses with a call-by-value parameter passing mechanism. These features are supported in a later implementation of SASL [b]. This latter implementation consists of three parts, a monitor which handles interaction with the user, a compiler which translates a program, the user submits to the system, into a code tree and an evaluator which evaluates the code tree by recursively evaluating its sub-trees. The evaluator does not suit our purposes, for it simulates the SASL machine at a higher level not allowing us to examine its progress step-by-step. Thus we constructed a new evaluator and interfaced it to the rest of the SASL system. This has enabled us to obtain a full SASL system and experiment with a number of non-trivial programs.

Since SASL distinguishes between the different data types at run-time rather than at compile-time the machine has a "tagged" architecture. The memory of the machine consists of a number of cells each of which contains two data items, a head and a tail. In this implementation the management of the memory is left to S-algol. This facilitates the implementation effort and makes simulating the interaction of machines less painful. The machine's other components are a Stack (S) and three special

+ [appendix II]
++ [appendix II, line 1124]
registers. A Control (C) register, an Environment (E) register and a Dump (D) register.

The Control
C
INDEX=0

\[ \rightarrow \text{PLUS} \]

\[ \text{ID} \quad \text{"X"} \quad \text{ID} \quad \text{"Y"} \]

**figure 3.1 - code tree for the expression \( X + Y \)**

The C register points to the node of a code tree currently being evaluated (or interpreted) by the machine, such nodes contain instructions. Their sub-trees denote the operands of the instructions. The number of operands depends on the type of instruction. In figure 3.1 the code tree for the expression \( X + Y \) is shown.

\[ \rightarrow \text{PLUS} \]

\[ \text{ID} \quad \text{"X"} \quad \text{ID} \quad \text{"Y"} \]

**figure 3.2 - pre-order evaluation of \( X + Y \)**

The C register has also a sub-component IHDX which parameterises the action of the machine for that instruction depending on whether none, one or both of the
operands to the instruction are accessible on the Stack. This is necessary since a code tree is traversed (evaluated) in pre-order. The INDEX takes the integer values 0, 1, 2.

The Environment

![Diagram of environment state component]

**Figure 3.3 - The Environment state component**

The k register points to a linked list of name-value pairs, figure 3.3. The list is organised as a stack to reflect the nesting of environments. Thus the environment is a structure which keeps track of the names that are currently in scope and their associated values. Nested definitions result in nested environments.

Initially all names in the environment are associated with suspensions. A suspension is a data structure with two data fields. A CODE field and an ENV field. It represents a "frozen" computation which on demand of its value the machine carries out by initialising its C and k registers from the suspension. On termination the value obtained overwrites the CODE field of the suspension and
the ENV is used as a flag to indicate to subsequent accesses that it has been evaluated. Thus, if frozen code is ever evaluated, it is only evaluated once.

The Dump+

<table>
<thead>
<tr>
<th>NEXTC</th>
<th>ENVR</th>
<th>LASTD</th>
</tr>
</thead>
<tbody>
<tr>
<td>INDEX</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

figure 3.4 - the state component Dump

The D register points to linked list of nodes each of which is a data structure with three fields, see figure 3.4. Each node identifies a state of the machine to be restored when the evaluation of the code subtree currently pointed to by the C register is completed. Since code trees are traversed in pre-order the C register pointing to the node of the tree is saved in the NEXTC field and set to the node of the left sub-tree from where it can be restored.

The evaluation of some code sub-trees is carried out by extending the current environment with local definitions. This extension to the current environment has to be undone when control (the C register) returns to the father node of the sub-trees. Thus prior to extending it, the current environment is stored in the ENV field of the data structure. The third field LASTD is used to organise the + [app. II, l. 1031]
list as a stack. Note the NEXTC field has a sub-field INDEX which indicates on restoring the state from the Dump what action remains to be done for the instruction. For example it may require checking the type of the value of the operand on top of the stack. An empty Dump indicates termination of the whole program.

Output

Initially the C register of the machine points to the root node of the code tree which the compiler produces from the program source. The root node identifies a special instruction PRINT with the rest of the code tree as its operand. The execution of PRINT causes the machine to save a "print" state on the Dump and continue with the evaluation of the operand of PRINT. Restoring the print state from the Dump causes the machine to output to the world outside the object referenced by the top of the Stack. [app. II, 1. 2601]

A sequence of PRINT/EVALUATE actions can be performed with this mechanism which enables the machine to handle the case where a list is to be printed. The machine evaluates a component of the list, it prints it and then goes on to evaluate the next component, printing an infinite list is handled in the same way except that the end of the list is never reached.

In general a list is computed as follows. Initially none of the components of the list are computed. A data
structure (a suspension, see above) with all the information to generate the list is passed around instead. The components of the list are evaluated, so that part of the list is actually generated, when access to the components of the list is required. This occurs when a list's component is an operand to a strict instruction (e.g., arithmetic) or the whole list is operand to the PRINT instruction. This is known as lazy evaluation.

The "unfreezing" of computations is print-driven (ie. nothing is evaluated unless it contributes to the calculation of an object to be printed).

Instruction set

The operation of the machine on each instruction comprises the following five basic actions which manipulate the components of the machine [app. II, lines 1040-1094]

I. pushstack (item):
The object denoted by item is pushed onto the Stack. In fact a reference to the run-time object is pushed onto the Stack.

II. popstack:
The top element of the Stack is popped.

III. cont.state (code):
The C register is set to point to the code sub-tree denoted by code. The INDEX component of the C register is set to 0. This indicates that no operands are available on
the Stack for the instruction at the node of the sub-tree.

IV. save.state (i):

The contents of the C, E and D registers are saved on the Dump. The INDEX component of the C register is set to the value i. This is the value of INDEX when the state is restored from the Dump.

V. load.state:

The top node of the Dump, pointed to by the D register, is popped and its contents initialise the C, E and D registers. This can be thought of as coming back to a "continuation" [30] left behind. The top of the Stack is the value passed to the continuation. If this value is suspended then it is evaluated and the result overwrites the code field of the suspension. The env field is used as a flag to indicate that the suspension has been evaluated.

The component INDEX of the C register is also initialised from the corresponding sub-field of the Dump. This indicates the number of operands to be expected on the Stack, currently accessible.

INDEX is also used to implement step-by-step actions of the evaluation mechanism such as parameter binding, clause matching (in the TRYS instruction below) and list selection (in the APPLY instruction below).

Each following instruction is given by its mnemonic followed by the names which denote its operands (sub-
trees). The effect of each instruction on the current state of the machine is parameterised by the INDEX of the Control component of the state and it is described in terms of the five basic actions I-V.
ID name

INDEX = 0:
lookup name in the current E
push its value on the Stack
push an error value if not found
if the top of the Stack is a suspension
perform the following actions:

save.state(1)
set C to the code field of the suspension
set E to the env field of the suspension
cont.state(C)

Otherwise:
load.state

INDEX = 1:
If the Stack top is a suspension
perform the following actions:

save.state(1)
initialise C and E from the suspension
cont.state(C)

Otherwise:

overwrite name in current E
load.state
CONDITIONAL e, e1, e2

INDEX = 0:
    save.state (1)
    cont.state (e)

INDEX = 1:
If the top is the object true:
    pop the top element of the Stack
    cont.state (e1)
If the top is the object false:
    pop the top element of the Stack
    cont.state (e2)
Otherwise:
    Replace the top element of the Stack
    with an error value.
    load.state
The name $d$ denotes a linked list of names and code sub-trees which represent computations of their values, see figure 3.5.

extend current $E$ with the definitions from $d$
each name is bound to a suspension
CODE fields initialised with corresponding code sub-trees and $\texttt{BHV}$ fields initialised with the extended current $E$
cont.state ($e$)
A closure is the machine's representation of a function, see figure 3.6. The field FORM denotes the parameter of the function being defined. It can be a constant, a name or a template the actual parameter must match. The field BODY denotes the code sub-tree to be evaluated as a result of applying the function to an argument. The field EE denotes the define-time Environment which is the current Environment E. On applying the function to an argument the sub-tree body is evaluated in environment en possibly extended with the binding of the formal parameter to the actual parameter (argument).
figure 3.7 - list cell creation

claim a new list cell (see figure 3.7)
create a suspension from e1 and current E
and initialise the head data field of the cell
create a suspension from e2 and current E
and initialise with it the tail data field of the cell
push a reference to the cell onto the Stack
load.state

CHECKLIST e

INDEX = 0:
  save.state (1)
  cont.state (e)

INDEX = 1:
  if top of the Stack is a list perform the following:

  load.state
Otherwise:

pop the top element of the Stack
push onto the Stack an error value
load.state

\text{HD } e

\textbf{INDEX} = 0:

save.state (1)
cont.state (e)

\textbf{INDEX} = 1:

if the top of the Stack is a list:

pop the top element of the Stack
push the contents of the head onto the Stack
load.state

Otherwise:

pop the top of the Stack and push an error value
load.state

\text{TL } e

Perform the same actions as when applying the operator \text{HD} to a list except that the tail of the list is pushed onto the Stack instead of the head.


\textbf{APPLY} \( e_1, e_2 \)

\begin{enumerate}
\item \textbf{INDEX} = 0:
  \begin{itemize}
  \item save.state (1)
  \item cont.state (e)
  \end{itemize}
\item \textbf{INDEX} = 1:
  \begin{itemize}
  \item if the Stack top is a list or a \texttt{BASIC FUNCTION}:
    \begin{itemize}
    \item save.state (2)
    \item cont.state (e2)
    \end{itemize}
  \item if the Stack top is a closure:
    \begin{itemize}
    \item create a suspension from \( e_2 \) and \( E \)
    \item push it onto the Stack
    \item bind formal parameter to actual parameter
    \item (the top element of the Stack)
    \end{itemize}
  \end{itemize}
\end{enumerate}

Binding may involve a matching process and may fail to match formal parameter to actual or it extends the current \( E \) with the formal parameter and its associated suspended value. This is referred to as the \texttt{call-by-need} parameter passing mechanism. If the binding process is successful it returns a new Environment otherwise it generates an error value which becomes the value of the function application.
set E to the result of binding
set C to the body of the function closure
cont.state (C)

INDEX = 2:
if the second from the top of the Stack element
is a list:

select the ith element of the list, where the top
of the Stack denotes i and the second from the top
element denotes the list
load.state

The selection process may involve generating the part of
the list which is suspended in order to reach the ith
element, see figure 3.8 (a),(b),(c) and (d)

![Diagram](figure3.8)
the $i$th element is reached when the top of the Stack, used as list selector is the value 1

![Diagram](image)

(c) (d)

**Figure 3.8**

If the second from the top of the Stack element is a **BASIC FUNCTION**: [app. II, 1. 986]

replace the top two elements of the Stack with the result of applying the function to the object on top of the Stack.

load.state

Note that **BASIC FUNCTION** is a predicate which tests the type of the object it is being applied to and it returns a Truth-value **true** or **false** on top of the Stack.

The class of instructions referred to by the mnemonics **BINOP** and **UNOP** represent the following arithmetic, relational and logical operators, defined over the appropriate type of objects.
"+", "*", "-", "/", "rem", "~", "|", "&", "=", "<=", ">=", "<", ">", "^".

These are mapped by the compiler to machine instructions PLUS, TIMES, MINUS etc.

\[ \text{BINOP } e1, e2 \]

\[ \text{INDEX } = 0: \]
\[ \text{save.state } (1) \]
\[ \text{cont.state } (e1) \]

\[ \text{INDEX } = 1: \]
\[ \text{save.state } (2) \]
\[ \text{cont.state } (e2) \]

\[ \text{INDEX } = 2: \]
replace the top two elements of the Stack with the result of applying the instruction to these elements, the result is an error value when the type of the operands are not of type expected by the instruction.

load.state
INDEX = 0:
  save.state (1)
  cont.state (e)

INDEX = 1:
  replace the top element of the Stack
  with the result of applying the instruction
  to this element
  load.state

This completes the instruction set of the machine except for the instruction THYS, similar to MAP but the closure it constructs represents a function defined by more than one clause. The implementation of multiple definitional Clauses is rather involved, relying on giving different values to the INDEX subcomponent of the Control component, each identifying a "state" of the process which selects the representation of the function whose formal parameter matches the actual parameter the closure is being applied to. The effect of the instruction THYS is described in detail in [6]. [app. II, 1. 1626]

Introduction of parallelism

In chapter two we proposed the use of annotation marks which induce the compiler to produce parallel instructions. Executing a parallel instruction, such as PAK-PLUS+ for example, has the effect of the current machine switching to

+ [appendix II, line 2086]
a WAIT state. The code sub-trees (operands) and current Environment of the machine initialise the C and E registers of new assistant machines. On termination of its assistants the machine may resume its computation.

The machine described in the previous section needs to be modified so that several machines combine their effort in executing a program. Each machine now has an additional Destination register whose contents identify its father machine. A machine is identified as a slot which receives a result. On termination a machine sends its result to this slot.

Since new machines are initialised with the same Environment, it is possible for them to access a suspension simultaneously or a machine to access a suspension which is currently being "coerced" to the value it denotes by another machine. Simultaneous access to a suspension which has been "coerced" (or "unfrozen") to the value it represents, poses no problem since all accesses are read-only. Otherwise simultaneous access or access while the suspension is being "coerced" to its value does present a problem.* In order to prohibit the same evaluation being carried out by different machines only the first machine must be allowed exclusive access. This saves unnecessary work being done. For this reason a suspension now has an extra "lock" field which is set by the machine which accesses it first and reset when it is overwritten. A machine which finds a suspension locked switches to a
Let us examine how deadlock may arise between, say, two evaluations A and B. This can only occur when they are data dependent upon each other.

A : Requires the value of a sub-expression let us name it X which is itself data dependent on the sub-expression Y

B : Requires the value of the sub-expression Y which is data dependent on the sub-expression X

In SASL this arises from certain definitions of the form

\[ X = ... Y ... \]
\[ Y = ... X ... \]

for example

\[ X = 1+Y \]
\[ Y = 1-X \]

which give the equation \( X = 2-X \) satisfied by the object undefined.

Note however the equation \( X = 1:X \) admits a solution, the infinite list \( 1:1:1:... \) and undefined is not a solution.

Thus deadlock only arises when a SASL program denotes the object undefined.
LOCKED state until it is evaluated.

The operation of the machine is extended by the following two actions: [app. II, lines 2352-2513]

**VI. spawn (code, env, slot):**

This action is invoked when a machine meets a parallel instruction. It causes a new assistant machine to be initialised by a code sub-tree denoted by code and by an Environment denoted by env. The slot initialises the Destination register of the machine. It denotes a place on the Stack of its father machine.

**VII. kill (machine id):**

This action is invoked on two occasions. Firstly, it is invoked by a machine which terminates its operation normally. Secondly, it is invoked by a father machine which no longer requires the result of the computation carried out by its child machine. The identity of the child machine is denoted by machine_id. We can think of the father machine sending a kill signal to its child. The kill signal is propagated by the child to all of its children and so on.

The effect of a parallel instruction is described below. The cases for the instructions PAR-OR and PAR-AND are treated as special ones since they are more powerful than the corresponding sequential ones.

\[ \text{PAK-BINOP } e_1, e_2 \]

+ [appendix II, line 2133]
PAR-BINOP

spawn(e1, ENV, slot1)
spawn(e2, ENV, slot2)
switch to WAIT state

The top two places on the Stack are reserved as slots to receive the results from the evaluation of the operands to the parallel instruction. A machine in WAIT state checks its Stack for results from its children, it then applies the instruction to these results. The machine resumes its progress by invoking the load.state action.

PAR-OK e1, e2

spawn (e1, ENV, slot1)
spawn (e2, ENV, slot2)
switch to WAIT state
WAIT state:

if top or second from the top element is the object true:

    kill (child)
    pop the two top Stack elements
    push onto the Stack the object true
    load.state

if one slot is the object false and the other is an error value:

    push onto the Stack an error value
    load.state

if both slots contain error values:

    push onto the Stack an error value
    load.state

if both slots contain the object false

    pop the Stack twice
    push this object onto the Stack
    load.state

Otherwise:

    remain in WAIT state

Similarly for the other parallel instructions PAR-AND and PAR-CONDITIONAL etc. [app. II, l. 1180]
Error Handling (app. II, 1. 2597)

The sequential (lazy) evaluator whenever it detects an error it terminates its progress and prints it as the value of the program. This is represented by the evaluated (or partially evaluated) code sub-tree. The node contains the instruction and the branches point to its evaluated operand(s), as shown in the example above. Since the control of the parallel evaluator is distributed this error value is treated as any other value. The corresponding task sends it to its father task this to its father and so on until the top task is reached which reports a partially evaluated code sub-tree (built bottom-up).

The partially evaluated code tree represents a trace of the computation carried out. The trace can be suppressed by having each task just propagating the smallest sub-tree (the error value) so that the error value climbs the tree of tasks unmodified.
CHAPTER FOUR

A model of parallelism

The parallel evaluator described in chapter three decomposes the evaluation of a program into a tree of tasks. The execution of a parallel instruction causes the current task to switch to a wait state until the operands of the instruction are available. These are to be evaluated as newly created tasks. In the implementation a new task is created by the primitive action spawn.

If a snapshot is taken at the parallel evaluation of a program the overall state is a composite of "smaller" states which form a tree, see figure 4.1. A node identifies a task in a particular state. ACTIVE states indicate the tasks are being processed, WAIT states identify tasks waiting for results of other tasks. A task is in a LOCKED state when it requires the value of a common suspension currently being "unfrozen" by another task. It remains in LOCKED state until the suspension is overwritten with the value it represents. The possibility of
interference of the above kind between tasks where a task becomes (dynamically) dependent on the value of another task other than the direct father/son dependency suggests that there is a graph of tasks and not just a tree. The broken line in figure 4.1 indicates that temporary data-dependencies arise between tasks, when the dependencies are resolved the related tasks still continue in existence. Unbroken lines show the flow of values which are obtained with the completion of tasks.

Conflict in the form of simultaneous access to the value of a suspension is a consequence of the efficient implementation of lazy evaluation in the environments model of computation (SECD type implementation). This is the technique by which non-strict functions and infinite lists are supported. In the graph reduction model of computation conflict would also arise between tasks due to "sharing" parts of the graph. Evaluating a shared part of the graph is felt simultaneously by all other references to it. On the contrary, string reduction gets round this problem by duplicating effort on common parts.

![Diagram](image)

**Figure 4.2** - ACTIVE tasks are associated with evaluators a, b, c
ACTIVE tasks are associated with the loci of control of evaluators which process them, shown by arrows in figure 4.2. In order to model the behaviour of a multi-processor machine we must take into account the following two observations.

![Diagram](image)

**Figure 4.3 - Tasks exceed evaluators**

First, as active tasks are being processed they generate new tasks. The number of created tasks, for a program of modest size, soon overwhelms the number of evaluators, see figure 4.3.

Second, the assignment of tasks to evaluators may involve considerable communication overheads. The above observations suggest that active tasks should not necessarily receive the attention of evaluators as soon as they are created. Thus in the model an assistant
evaluator to the current evaluator is employed only after a certain amount of "time" has elapsed (see below).

\[ a \rightarrow (a) \]
\[ b \rightarrow (b) \]
\[ a \rightarrow (a) \]

figure 4.4 – each evaluator simulates the parallel evaluation of tasks

a: main evaluator, b: assistant

In the absence of assistant evaluators the locus of control of the current evaluator traces a bottom first leftmost path over the tree of tasks, see figures 4.4 (a),(b) and (c). Thus each evaluator will attempt to simulate the parallel evaluation of tasks it creates.

When an assistant evaluator to the current one is actually employed it is assigned the last task to be processed by the current evaluator. Further assistants similarly are assigned the next to last tasks. Thus assistant evaluators take load off the current evaluator. These may have the benefit of other assistants in the same fashion and so on.
The effect of a "parallel run" of a program in the model is measured by the amount of effort the initial evaluator exerts. This is the number of steps it goes through to evaluate its input program. A step is equivalent to the execution of one instruction, as described in chapter three. Thus in the model the locus of control of the initial evaluator is associated with a count of the number of instructions it performs. Also a count of the number of lock steps which occurred during its progress as well as the total number of lock steps is noted for each run.

![Diagram](image)

**figure 4.5**

In the case shown in figure 4.5 where the main evaluator has come back to a task assigned to an assistant evaluator which has not completed it yet, it is assumed from this moment onwards that the effort of the assistant evaluator counts as if it was exerted by the main evaluator.

**When to spawn**

It has been mentioned that an assistant evaluator is
employed after some "time" has elapsed. During this time several (or no) new tasks may have been created pending processing. Time is related to the amount of work the evaluator performs. Thus its locus of control is associated with a clock which registers its effort. The clock is set to a certain threshold which, when it gets exceeded, causes the initiation of an assistant evaluator. This occurs every time the threshold is exceeded providing there are tasks to be processed. Time has been measured in three different ways.

First, as the number of instructions executed.

Second, as the number of COLON instructions (list cell creations).

Third, as the number of APPLY instructions (function applications).

In order to make this quantity ("time") relative to the evaluation of each program experimented with, a threshold is computed as a percentage of the total number of instructions performed under sequential evaluation where a single evaluator is employed.

Note that the delay a threshold imposes is finite so that the correct result of parallel operators such as "$\mid\#$" (PAR-OR) and "$\&\#$" (PAR-AND) is computed. If the evaluation of one of the operands diverges then eventually the threshold which prohibits the spawning of the task for the
other operand will be exceeded. This would cause the second operand task to be processed. If its value is true for OR or false for AND then the application of the parallel operator will return this value as its result.

Each program is evaluated in the model under different strategies of spawning where a strategy is determined by the particular threshold imposed. All strategies are bounded by two extremum cases.

The **Totally sequential** case where only a single evaluator is employed. This evaluator simulates the parallel evaluation of all tasks. So the partial order of tasks represented by the graph in figure 4.4 (a) is flattened to a total order.

The **Maximally parallel** case where a new evaluator is employed as soon as a task is created. Between these two strategies there is a spectrum of strategies which result in imposing an order on tasks otherwise unordered.

A series of experiments is performed for each program under different strategies. The outcome of each experiment for each program apart from its result provides the following information.

- The number of steps performed by the initial evaluator, as a percentage optimisation over the number of steps under totally sequential strategy.
- The number of lock steps of the initial evaluator.
These constitute actual delay in the overall evaluation.

The total number of lock steps indicating the amount of interference between evaluators. The performance in each experiment is plotted against the corresponding strategy.

Sample points taken at regular intervals during the evaluation which show the number of tasks being processed at each sample point. The profile of an evaluation is presented as a histogram. The results of experiments appear in chapter six.

Simulation

![Diagram of simulation process]

**figure 4.6**

In the absence of real concurrency the behaviour of the model of parallelism is simulated by a program which executes sequentially. Its locus of control (S-ALGOL's) timeshares over the tasks so that each task is processed for a timeslice equivalent to the execution of one instruction.

The interaction between evaluations of tasks is
modelled at instruction execution level. Modelling at sub-instruction level would be required to examine storage management problems for example. Such a simulation is reported in [31]. Figure 4.6 shows how parallelism is achieved. All tasks are arranged in a ring structure with the processor going round the ring giving each task a step equivalent to one instruction. Tasks in wait state just examine their Stack slots to see if their assistant tasks have produced any results. Tasks in locked states examine the field "lock" of the suspension. Pending tasks which have not "fired" yet are ignored.

The action of a task killing its sub-task as it discovers it does not need its result any longer, is assumed to occur instantaneously before any other task changes state. This is a rather ideal situation since the problem of identifying irrelevant tasks and terminating them in order to recover the portion of resources allocated to them is not a trivial problem [20,32,33]. The main problem is that of "chasing" where if the number of newly generated tasks, sub-tasks of a killed task, which receive the attention of processors, grows faster than the rate of killing them then this can result in the machine being taken over by irrelevant tasks. This is analogous to the case where a garbage collection process runs out of space itself while trying to recover unwanted space in a sequential machine.

The simulation works at a level above the problems of
resource allocation that a real multi-processor machine would have to deal with. Here the main idea is to discover the amount of parallelism "logically" present which can be exploited. The simulation does not answer the problem of whether such parallelism can be "physically" realised.

From the point of view of a parallel architecture the ring suggests the arrangement shown in figure 4.7.

![Diagram](figure 4.7)

The machine consists of a pool of processors and a pool of tasks. A task has some portion of the total memory engaged. The fact that the run time structure is highly interleaved suggests that there must be a globally referenced memory divided into blocks. As a task is being processed it generates more tasks which can be processed by the current processor or other processors. The proposals for architectures [31,39,40] take up this problem more fully.
CHAPTER FIVE

Parallel Programs

In this chapter we examine a number of SASL programs with the purpose of identifying evaluations that can be carried out in parallel. In some cases the original program must be transformed or even replaced by a more parallel program.

An expression represented as a graph of data dependencies shows the evaluations that can be carried out in parallel. Evaluations are ordered by the data dependencies that arise in their evaluations. A data dependency indicates that computing the value of an expression requires that of another expression.

The complexity of evaluations is important in deciding the grain of parallelism [34]. This is a criterion by which we consider, for example, the operation of multiplying two matrices as appropriate for organising it in parallel, whereas we consider the multiplication of two integers not appropriate because the grain of parallelism in this instance is too fine. Consider the program for computing the exponentiation function
\[ \text{expo } x \ 1 = x \]
\[ \text{expo } x \ n = x \ * \ \text{expo } x \ (\ n-1 \ ) \]

\[ \text{figure 5.1 - program graph of expo} \]

Its graph representation, shown in figure 5.1, indicates sub-expressions \( x \) and \( \text{expo } (n-1) \) may be evaluated in parallel. The complexity of the evaluations though suggests rather unbalanced evaluations. This means there is relatively little amount of work to be done in parallel.

\[ \text{splitexpo } x \ n \]
\[ \text{splitexpo } x \ (n/2) \quad \text{splitexpo } x \ (n-(n/2)) \]

\[ \text{figure 5.2 - program graph of splitexpo} \]

A transformation of the program produces a balanced split exponentiation function, see figure 5.2, defined as

\[ \text{splitexpo } x \ 1 = x \]
\[ \text{splitexpo } x \ n = \text{splitexpo } x \ (n/2) \ * \]
\[ \text{splitexpo } x \ (n-(n/2)) \]
Now we can interpret the primitive operator \( * \) as being parallel. For that purpose we introduce an annotation symbol \( \# \) which directs the compiler to generate a parallel instruction for the benefit of the evaluator. Parallel instructions cause an evaluation path to split into parallel paths.

**Examples from graph theory**

![Directed Graph](image)

**Figure 5.3 - A directed graph**

Graphs model many real life situations so graph manipulating programs are interesting cases to examine. In particular we will examine graphs of relationships.

A directed graph \( G \) consists of a finite number of vertices and arcs labelled by a direction. We choose to name vertices by integers. The directed graph shown in figure 5.3 has vertices 1, 2, 3 and arcs 12, 13, 22, 23. Arrows indicate the direction of each arc. We define a function \( G \) to represent the graph.
\( G_0 = 3 \) || the size of graph in vertices
\( G_1 = 2,3 \)
\( G_2 = 2,3 \)
\( G_3 = () \)

The function \( G \) is passed as a parameter to graph manipulation functions.

Example 1

To compute the reachability relation of a graph \( G \), by following the outgoing arcs from each vertex.

program

\[
\text{Rel } G = \\
\text{FOR } i \ (G i) \ \text{reach} \\
\text{where} \\
\text{reach } i = i,'to',\text{extend } () \ i \ , \ \text{nl} \\
\text{extend sofar } i = \text{MEMBER } \text{sofar } i \rightarrow () \\
\text{UNION } \text{arcs } \text{succs} \\
\text{where} \\
\text{arcs } = G \ i \\
\text{succs } = \text{MAPUNITE } (\text{extend } (i: \text{sofar})) \ \text{arcs}
\]

The standard functions \text{FOR}, \text{MAPUNITE}, \text{UNION} and \text{MEMBER} are defined by the following Clauses
Sets are represented by lists. The output of the function \( \text{FOR} \) is a list. This represents the reachability relation (can be thought of as a new graph) for the input graph. The components of the list are computed sequentially as the list is being printed.

Parallelism manifests here as parallel evaluation of the list's components. This is effected by defining a function \( \text{SPLITFOR} \) which computes the list as a balanced tree (represented as a list of lists) and then flattens the tree into a linear list by the \( \text{APPEND} \) function.

\[
\text{SPLITFOR } a \ a \ b \ f = a > b \rightarrow ()
\]
\[
f a : \text{FOR} \ (a+1) \ b \ f
\]
\[
\text{MAPUNITE } f () = ()
\]
\[
\text{MAPUNITE } f (a:x) = \text{UNION} \ (f a) \ (\text{MAPUNITE} \ f \ x)
\]
\[
\text{MEMBER } () \ a = \text{false}
\]
\[
\text{MEMBER} \ (a:x) \ a = \text{true}
\]
\[
\text{MEMBER} \ (a:x) \ b = \text{MEMBER} \ x \ b
\]
\[
\text{UNION } () \ y = y
\]
\[
\text{UNION} \ (a:x) \ y = \text{MEMBER} \ y \ a \rightarrow \text{UNION} \ y \ x
\]
\[
a : \text{UNION} \ x \ y
\]

\[
\text{APPEND } h \ t = h ++ t
\]
To transform the function SPLITFOR to a parallel function we use the function STRICT which simulates simultaneous call-by-value on the operands of append, this is defined as call-by-parallel. Note there is always a choice to be made concerning the grain of parallelism which selects a certain function to be transformed into a parallel one. This involves apart from the structure of the corresponding flow graph knowledge of the complexity of the function. Whether this is left to the user to decide or for the system to cope with automatically is an open question. For example the function MEMBER which scans a list could also be chosen for parallel transformation.

Example 2

A directed graph G is called cyclic if there exists a vertex which can reach itself. To test whether a given graph is cyclic we use the function extend defined in the previous example.

```plaintext
program cyclic G = cycleat 1

where

cycleat i = i > G 0 -> false

MEMBER path |# cycleat (i+1)

where

path = extend () i
```

The evaluations of sub-expressions

```
MEMBER path i cycleat (i+1)
```
operands to the parallel operator \(\#\) (parallel-or) can be carried out simultaneously. The relative complexity of the evaluations suggests that they are unbalanced. So we must transform the function cycleat so that the looping it entails is unfolded in a tree structure with the operator \(\#\) at the nodes.

\[
\text{cycleat } i \ i = \text{MEMBER } \text{path } i
\]

\[
\text{where}
\]

\[
\text{path} = \text{extend } () \ i
\]

\[
\text{cycleat } i \ j = \text{cycleat } i \ \text{mid } \# \ \text{cycleat } (\text{mid}+1) \ j
\]

\[
\text{where}
\]

\[
\text{mid} = (i+j)/2
\]

**Example 3**

Modify the previous program to compute the vertex at which the cycle starts. We can modify the function cycleat to return the name of the vertex instead of true and the empty list instead of false.

**program**

\[
\text{cycleat } i \ i = \text{MEMBER } \text{path } i -> i ; ()
\]

\[
\text{cycleat } i \ j = \text{or } (\text{cycleat } i \ \text{mid})
\]

\[
(\text{cycleat } (\text{mid}+1) \ j)
\]

\[
\text{or left right} = \text{right} = () -> \# \text{ left}
\]

\[
\text{right}
\]

To effect parallelism we define a parallel conditional operator \(\rightarrow\#\) which evaluates the predicate expression and
the left alternative in parallel. Note the because of the
data dependency of the predicate to the right alternative
we do not need a full parallel conditional. A second
annotation mark would be required to define such an
operator. Termination of the predicate with the value
false causes the evaluation of the left alternative to be
forcibly terminated, if it is still going on, as
irrelevant.

Example 4

A vertex of a directed graph is called terminal if a
directed cycle cannot be reached from it. If a graph is not
cyclic the set of terminal vertices consists of all the
vertices of the graph.

Compute the set of terminal vertices of a directed graph.

program

    terminals G =
    FILTER term (COUNT 1 (G 0))
    where
    term i = ~nont i
    nont i = OR (MEMBER path i : MAP nont path)
    OR () = false
    OR (a : x) = a | OR x

we use the function "extend" from example 1. The function
COUNT computes the list 1, 2, ...(G 0) which is filtered
to leave in only those components (vertices) which satisfy
the predicate term. We choose to parallelise the function FILTER. We replace the list 1, 2, ..., (n 0) by introducing an extra parameter in FILTER and apply the split transformation to it.

\[
\text{FILTER } p \ n \ n = p \ n \rightarrow n,
\]

\[
()\]

\[
\text{FILTER } p \ n \ m = \text{APPEND} (\text{FILTER } p \ n \ mid) \ (\text{FILTER } p \ (\text{mid+1}) \ m)
\]

where

\[
\text{mid} = (n+m)/2
\]

now APPEND is prefixed by STRICT as in example 1. Further parallelism is possible from the function OR which scans a list looking for the object true as soon as it finds this object it returns it as its result, otherwise it returns the object false. The parallel OR function is defined as follows

\[
\text{OR} () = \text{false}
\]

\[
\text{OR} (a:x) = a \ |\# \ \text{OR} x
\]

note since the sequential function OR does not need to evaluate all the components of the list, only as far as the first true, the parallel OR function involves evaluating components in anticipation that their value might be needed.

Example 5

In a directed graph when a vertex v has an arc to a
vertex \( u \) then the vertex \( u \) is called the **successor** of \( v \) and \( v \) is called the **predecessor** of \( u \). For some vertex \( i \) the **minimal transition pair** with \( i \) as the initial vertex is the smallest pair of sets \( M \) and \( N \) such that

- vertex \( i \) is a member of \( M \)
- all successors of \( M \) are members of \( N \)
- all predecessors of \( N \) are members of \( M \)

The following program computes the sets \( M \) and \( N \) for a given graph \( g \) which satisfy the above conditions.

\[ \text{mtpair } g \]

where

\[ MN \text{ mset nset} = c1 \land c2 \rightarrow \text{mset, nset} \]

\[ MN \text{ (c1 } \rightarrow \text{ mset ; mset1)} \]
\[ (c2 \rightarrow \text{ nset ; nset1} \]

where

- \( c1 = \text{SUBSET succs mset} \)
- \( c2 = \text{SUBSET preds nset} \)
- \( \text{succs} = \text{MAPUNITE succ mset} \)
- \( \text{preds} = \text{MAPUNITE pred nset} \)
- \( \text{nset1} = \text{UNION nset succs} \)
- \( \text{mset1} = \text{UNION mset preds} \)

\[ \text{succ} \ v = \ g \ v \]
\[ \text{pred} \ v = \text{FILTER (arc v) (COUNT 1 (g 0))} \]
\[ \text{arc} \ i \ i = \text{false} \]
\[ \text{arc} \ i \ j = \text{member (succ} \ j) \ i \]
\[ \text{COUNT} \ a \ b = a > b \rightarrow () \]
\[ a : \text{COUNT} (a+1) \ b \]
The function COUNT computes the list 1, 2, ..., n which is the list of vertices of the input graph g.

An undirected graph, shown in figure 5.4, is represented here with a double arc.

![Figure 5.4 - An undirected connected graph](image)

such a graph is called connected if every vertex is reachable from any other. An undirected connected graph is called bipartite if its vertices can be partitioned in two sets M and N such that no edge (a double arc) joins two vertices of the same set. To solve the problem whether a given graph is bipartite can be programmed as follows.

Let i be an initial vertex, say 1. We can use the function MN, defined above, to assign the vertices to two sets M and N such that vertices joined by an edge are assigned to different sets. As the graph is connected all vertices will be assigned to at least one set. The graph is not bipartite if a vertex has been assigned to both sets.
program

bipartite \( g = \text{empty} (\text{INTERSECTION } M \ N) \)

where

\( \text{empty} () = \text{true} \)

\( \text{empty } s = \text{false} \)

\( M,N = MN (1,) (\text{succ} 1) \)

Note since vertices are joined by double arcs there is no need for the function \( \text{pred} \), just use \( \text{succ} \). The empty set is represented as the empty list \( () \). The function \( \text{INTERSECTION} \) computes the denoted set operation. In order to transform \( mtpair \) into a parallel program the operator \( \& \) is replaced by the parallel operator \( \&\& \) so that subexpressions \( c_1 \) and \( c_2 \) are evaluated in parallel. Since \( \&\& \) is strict in only one of its operands (see the \( \text{PAK-AND} \) instruction in chapter three), termination of one of the evaluations, say \( c_1 \) for example, giving \( \text{false} \) causes the termination of the evaluation of \( c_2 \) as irrelevant.

![Diagram](image)

**Figure 5.5 - Program graph of "MN"**

Note that it is possible that the value of \( c_2 \), for example, will be required by the evaluation of the expression
the graph of "MN" in figure 5.5 indicates that the latter evaluation is data-dependent on the evaluation, characterised by the operator && as speculative. This suggests that both the values of $c_1$ and $c_2$ must be found before the operator && is applied. So the sub-expression $c_1 \& c_2$

in the sequential program is replaced by

\[
\text{STRICT and } c_1 \lor c_2
\]

where the function "and" is defined by

\[
\text{and } x \lor y = x \& y
\]

Example 6

To test whether a function contains a zero in a given interval within a given accuracy criterion (the local version of SASL does not cope, at present, with real numbers but the program will work on a variety of "scaled" integer functions).
The method of solution is to divide the given interval into two sub-intervals and search for a zero of the function in the sub-interval which indicates the function crosses the x-axis. If neither sub-interval indicates this condition they are searched left to right by being subdivided further.

program

Root f x y e = x-y < 2*e ->

  negsign x y -> 'root is " , mid
  'no root found",
  negsign x mid -> left
  negsign mid y -> right
  ONEOF left right
  where
  negsign a b = f.a * f b < 0
  left = Root f x mid e
  right = Root f mid y
  mid = (x+y)/2

ONEOF m n = isnroot m -> n

  m

isnroot (mesg:x) = x = ()

Parallelism here manifests as splitting the interval and pursuing the test on each sub-interval in parallel. Success on one of the sub-intervals renders the search in the other as irrelevant (if one is looking for just one root).
Again the full parallel conditional operator was not needed. Only the condition and left alternative need be evaluated in parallel. Note that for the particular case where the pattern of searches followed by the sequential program is optimal, this occurs when the sequential program never takes up a right half interval, the introduction of parallelism does not improve the performance. In general though we can safely assume this will not be the case. Note also that as soon as a path hits success this is detected by the immediate application of ONEOF and reports it to the outer application of itself so that the answer reaches the top of the tree causing termination of search paths on its way. This is effected by replacing \( \rightarrow \) by the parallel operator \( \rightarrow # \) in the body of the function ONEOF (see example 3).

Example 7

The program to compute the moves of discs which solve the towers of Hanoi.

```
program

Hanoi 0 (a,b,c) = ()
Hanoi n (a,b,c) = Hanoi (n-1) (a,c,b),
move,
Hanoi (n-1) ( b,a,c )
where
move = 'disc "a," to"c'
```

To transform the function Hanoi into a parallel
function we just replace the two occurrences of comma by a function \text{comm2} and use \text{STRICT} to force call-by-parallel on the parameters of the function \text{comm2}.

\[
\text{Hanoi } n \ (a,b,c) = \text{STRICT } \text{comm2} \ l \ r \\
\text{where} \\
\text{comm2} \ l \ r = l, \text{move}, r \\
l = \text{Hanoi } (n-1) \ (a,c,b) \\
r = \text{Hanoi } (n-1) \ (b,a,c)
\]

Note that no transformation of the program to enhance parallelism is required since the evaluation of subexpressions \(l\) and \(r\) are of the same complexity.

\textbf{Example 8}

To compute the matrix product of two matrices. In order to present a clearer program let us assume the matrices are square of dimension \(n\), power of 2. A matrix is represented as a list of lists in row order. For example the expression

\[
((1,0),(0,1))
\]

represents the unit square matrix of order 2. We define the product in terms of inner product operations between vectors. A row or a column of a matrix constitutes a vector. The inner product function \(\text{IP}\) is defined by the following Clauses

\[
\text{IP } () () = 0 \\
\text{IP } (r : x) (c : y) = r \times c + \text{IP } x \ y
\]
a matrix is transposed by the function transpose below

\[
\text{transpose } M = \text{map } \text{hd } M : \text{transpose (map } \text{tl } M) \\
\text{hd } (a : x) = a \\
\text{tl } (a : x) = x
\]

The \text{ith} row of the product matrix is formed by taking the inner product of the \text{ith} row of matrix \( M \) with all the columns of matrix \( N \).

\begin{verbatim}
program

\text{multiply } M N = \text{mult } M (\text{transpose } N) \\
\text{mult } () \text{ cols} = () \\
\text{mult } (r : \text{ rows}) \text{ cols} = \text{new } r : \text{ mult } \text{ rows cols} \\
\text{where} \\
\text{new } row = \text{MAP (IP } \text{ row)} \text{ cols}
\end{verbatim}

We identify parallel evaluations at the level (grain) of function \text{mult} where the operands of : can be evaluated in parallel. Similarly at the inner level of \text{MAP} used by the function \text{new} and finally at the level of function \text{IP}.

The infix operator : can be replaced by a function \text{cons} and then we can use the function \text{STRICT} to force call-by-value on the actual parameters of \text{cons}. The complexity of the function \text{new} and more obviously of \text{IP} with respect to the complexity of the whole program suggests that we only consider parallelism at the level of the function \text{mult}. Note that had we decided to consider parallel evaluations, say at the level of function \text{IP}, we would need to transform
this function in order to balance the tree of evaluations that the parallel + operator gives rise to.

The same criticism applies in introducing parallelism at the level of function mult whose parallel balanced version may be defined as follows

\[
\text{mult rows cols} = \text{split 1 (LENGTH rows)}
\]

\[
\text{where}
\]

\[
\text{split i i} = \text{new (rows i)},
\]

\[
\text{split i j} = \text{STRICT APPEND}
\]

\[
\text{(split i mid)}
\]

\[
\text{(split (mid+1) j)}
\]

Now the rows of the product matrix are computed in parallel. Closer examination of the algorithm shows that each such evaluation requires access to the column matrix cols. This implies the evaluations cannot proceed independently of each other. In order to obtain an effectively parallel program for matrix multiplication we therefore look for a different algorithm, in fact the function split above provides the idea. The computation of an element is given by the formula

\[
c_{ij} = a_{i1} b_{1j} + a_{i2} b_{2j} + a_{i3} b_{3j} + \ldots
\]

Let us consider multiplying matrices A and B obtaining matrix C, all of dimension n, a power of 2.
this function in order to balance the tree of evaluations that the parallel \(+\) operator gives rise to.

The same criticism applies in introducing parallelism at the level of function \(\text{mult}\) whose parallel balanced version may be defined as follows

\[
\text{mult rows cols} = \text{split 1 (LENGTH rows)}
\]

\[
\text{where}
\]

\[
\text{split i i} = \text{new (rows i),}
\]

\[
\text{split i j} = \text{STRICT APPEND (split i mid)}
\]

\[
\text{(split (mid+1) j)}
\]

Now the rows of the product matrix are computed in parallel. Closer examination of the algorithm shows that each such evaluation requires access to the column matrix \(\text{cols}\). This implies the evaluations cannot proceed independently of each other. In order to obtain an effectively parallel program for matrix multiplication we therefore look for a different algorithm, in fact the function \(\text{split}\) above provides the idea. The computation of an element is given by the formula

\[
c = a \ b + a \ b + a \ b + \ldots
\]

Let us consider multiplying matrices \(A\) and \(B\) obtaining matrix \(C\), all of dimension \(n\), a power of 2.
we can divide matrices $A$ and $B$ so that they form $(2 \times 2)$ matrices whose elements are $(n/2) \times (n/2)$ matrices. An element $C_{ij}$ of the product matrix is computed using the same equation as above, for example

$$C_{11} = A_{11}B_{11} + A_{12}B_{21}$$

but the operands of addition and multiplication are matrices. The equation indicates the two multiplications can be carried out in parallel. Note each multiplication between the matrices $A_{ij}$ and $B_{ij}$, of dimension $n/2$, will give rise to parallel evaluations of matrices of dimension $n/4$ and so on until multiplication of atomic operands is reached.

The computation this algorithm implies recursively subdivides into non-trivial independent evaluations. The new algorithm for matrix multiplication is yet another example of the approach to problem solving known as the Divide-and-Conquer method. In fact we have already encountered many examples of this method when the functions splitexpo, SPLITFOR, cycleat were defined. Programs implementing this type of algorithm are ideally suitable for parallel evaluation, since their evaluation splits evenly into sub-evaluations. These can be carried out in parallel.

In order to change the representation of a $(n \times n)$ matrix $A$ so that $A_{ij}$ is not an integer but a $(n/2) \times (n/2)$ matrix we define a function make2 to do the conversion.
make2 A n = G

where

G 1 1 = F

\[ \text{where } F_{i,j} = A_{i,j} \]

G 1 2 = F

\[ \text{where } F_{i,j} = A_{i,(j+\text{offset})} \]

G 2 1 = F

\[ \text{where } F_{i,j} = A_{(i+\text{offset}),j} \]

G 2 2 = F

\[ \text{where } F_{i,j} = A_{(i+\text{offset}),(j+\text{offset})} \]

offset = n/2

Since the results of operations are square matrices printed as lists of lists, we define a function MATRIX which produces a square matrix

\[ \text{MATRIX } n \ e = \text{FOR } 1 \ n \ r \]

\[ \text{where } \]

\[ r_{i} = \text{FOR } 1 \ n \ c \]

\[ \text{where } \]

\[ c_{j} = e_{i,j} \]

The addition of matrices of dimension k, represented as 2X2 matrices with elements matrices of dimension k/2 is defined by function SUM as follows

\[ \text{SUM } F \ G \ 1 = F + G \]

\[ \text{SUM } F \ G \ k = \text{MATRIX } 2 \ e \]

\[ \text{where } \]

\[ e_{i,j} = \text{SUM } (F_{i,j}) (G_{i,j}) (k/2) \]
The multiplication of 2x2 matrices is defined in terms of the function `MATRIX` as follows:

\[
\text{mult2 } A \ B = \text{MATRIX } 2 ((A_{i1}B_{1j})+(A_{i2}B_{2j}))
\]

Now we can define matrix multiplication anew in terms of the functions `mult2`, `make2`, `MATRIX` and `SUM` program:

\[
\text{multiply } A \ B \ 2 = \text{mult2 } A \ B
\]
\[
\text{multiply } A \ B \ n = \text{mult } A_2 \ B_2 \ (n/2)
\]
\[
\text{where}
\]
\[
A_2 = \text{make2 } A \ n
\]
\[
B_2 = \text{make2 } B \ n
\]

\[
\text{mult } M \ N \ k = \text{MATRIX } 2 \ e
\]
\[
\text{where}
\]
\[
e_{ij} = \text{SUM} (\text{multiply } (M_{i1}) (N_{1j}) \ k)
\]
\[
(\text{multiply } (M_{i2}) (M_{2j}) \ k)
\]
\[
k.
\]

We identify parallelism at the level of function `MATRIX` where the elements of the matrix can be evaluated in parallel. This is effected by transforming `FOR` into a parallel function. Note that since the first parameter of `MATRIX` is 2 the function `FOR` produces a 2-list so that there is no need to transform `FOR` to the function `SPLITFOR`, we have met this function in example 1.

Parallelism is also identified at the level of function `SUM` where the operands of the function `SUM` can be evaluated.
in parallel. We choose the level of the function SUM because the grain of the function MATRIX overwhelms the simulator, even for a small (8x8) matrix.

Note the bulk of the work is done at the level of the function SUM and although an element may be evaluated before another is it has to be printed in a particular order. To effect parallelism the function STRICT is used to perform call-by-parallel on the operands of SUM

\[ e_{i,j} = \text{STRICT SUM} \left( \text{multiply} \ (M_{i1}) \ (N_{1j}) \ k \right) \]
\[ \left( \text{multiply} \ (M_{i2}) \ (N_{2j}) \ k \right) \]

Example 9

To sort a list of integers in ascending order. There are a number of sorting algorithms [35]. We choose the sort by merge algorithm because it employs the Divide-and-Conquer technique. Other sorting methods such as quicksort do this also but are not considered here. Given a list of \( n \) numbers, split it into two sub-lists of \( n/2 \) and \( n+2/2 \) numbers and then merge the sorted sub-lists.
program

sort x = split 1 (LENGTH x)
where
split n n = x n,
split n m = STRICT merge
  (split n mid)
  (split (mid+1) m)
where
mid = (n+m)/2

merge () y = y
merge x () = x
merge (a : x) (b : y) = a <= b -> a : merge x (b : y)
                             b : merge (a : x) b

Note that all parallel evaluations require access to some element of the list x.

Example 10

To compute a relation from two relations. The relations are two tables of library information. One table gives the relation between books and authors and the other between borrowed books and names of borrowers. The relation to be computed is defined as "the list of authors whose books are lend to other authors".

A table is represented by a list of pairs. Each pair is represented by a 2-list. The list of book-author pairs is denoted by the parameter BAL and the book-borrower list of
pairs is denoted by the parameter $BBL$.

**Program**

```markdown
new_rel $BAL \ BBL = relation \ BBL$

where

$relation() = ()$

$relation(p:x) = rel p \rightarrow p 2:relation \ x$

$rel(bk,br) = AND(author\sim()=(),author\sim=br)$

where

$author = FIND \ BAL \ br$

$AND() = true$

$AND(a:x) = a \rightarrow AND \ x; \ false$

$FIND() \ item = ()$

$FIND((item,nm) : x) \ item = nm$

$FIND((bk,item) : x) \ item = bk$

$FIND(p : x) \ item = FIND \ x \ item$
```

The function $FIND$ searches the list of pairs for an item, if it finds the item contained in a pair it returns the related object. Both functions $relation$ and $FIND$ may be parallelised. We choose the grain of parallelism offered by the latter function which performs $FIND$ steps recursing on its first parameter. We transform the recursions into a tree whose terminals, left to right, are the unwound recursions.
relation \( x = \text{split} \ 1 \ (\text{LENGTH} \ x) \)

where

\[
\text{split} \ n \ n = \text{rel} \ (x \ n) \rightarrow x \ n \ 2,
\]

()  

\[
\text{split} \ n \ m = \text{APPEND} \ (\text{split} \ n \ \text{mid})
\]

(\text{split} \ (\text{mid}+1) \ n)

where

\[
\text{mid} = (n+m)/2
\]

By prefixing \text{APPEND} with the system function \text{STRICT} a parallel program is obtained.

Example 11

A partition of an integer \( n \) is a collection of positive integers whose sum is \( n \). The integers in the collections are called the parts of the partition. We do not impose any other restrictions on the partitions. Consider the partitions of the first three integers.

\[
\begin{align*}
1 \\
2 & 11 \\
3 & 21 & 12 & 111
\end{align*}
\]

We see that the partition of 3 is generated from those of 2 and 1 by extending (prefixing) the partitions of 2 with \( 1 = (3-2) \), obtaining 12 111 and of 1 with \( 2 = (3-1) \) obtaining 21 and finally of 0 which is empty (nullpart) with \( 3 = (3-0) \) to get 3.
program

nullpart = (),

part 0 = nullpart
part n = fora 1 n last

where
    last i = prefix i (part (n-i))
    prefix i () = ()
    prefix i (p : x) = p=nullpart=>(i,) : prefix i x
                        (i : p) : prefix i x

fora a b f = a > b -> ()
              f a ++ fora (a+1) b f

By paralleling function fora, the partitions may be generated in parallel.

fora a b f = a=b -> f a,

    STRICT APPEND (fora a mid f)
    (fora (mid+1) b f)

    where
    mid = (a+b)/2

Note that same partitions are recomputed, following the technique of [36] we modify the function part to be a memo function which remembers previously generated partitions.
partlist = MAP part (from 0)
part n = fora 1 n last
where
    last i = prefix i (partlist (n-i+1))

Example 12

To generate the permutations of set of integers. We take the solution given in the Sasl manual [4].

program

perms () = (),
perms x = f x
where
    f (a : y) = MAP (cons a) (perms y) ++
        g (y ++ (a,))
    g y = y = x -> ()
    f x

Similarly here replacing ++ by the function APPEND and using STRICT to effect simultaneous call-by-value of the parameters of APPEND, the evaluation path splits into parallel sub-paths. One path computes the permutations of a list of numbers where the first element is fixed. The other rotates the list and computes its permutations. Each sub-path follows the same split pattern.

let us consider the same algorithm expressed somewhat differently so that parallel paths are of the same
complexity, where the loop defined by g has been taken out and externalised.

\[
\text{perms } x = \text{MAP } f \ x
\]

where

\[
f \ a = \text{MAP} \ (\text{cons } a) \ (\text{perms} \ (\text{diff} \ x \ a))
\]

diff () a = ()
diff (a : x) a = x
diff (b : x) a = diff x a

we replace MAP by the function "split"

\[
\text{perms } x = \text{split} 1 \ (\text{LENGTH} \ x)
\]

where

\[
\text{split} \ n \ n = f \ (x \ n),
\]

\[
\text{split} \ n \ m = \text{STRICT APPEND}
\]

\[
\text{(split} \ n \ \text{mid})
\]

\[
\text{(split} \ (\text{mid}+1) \ m)
\]

where

\[
\text{mid} = (n+m)/2
\]

**Example 13**

The queens problem where the queens are to be placed on an (n×n) board in such a way that none checks any other. We use the solution of [4] where the board is represented by a list. The components of the list represent the columns of the board. Each component is an integer and its value represents the row of the board.
complexity, where the loop defined by \( g \) has been taken out and externalised.

\[
\text{perms } x = \text{MAP } f \ x \\
\text{where} \\
f \ a = \text{MAP} \ (\text{cons } a) \ (\text{perms} \ (\text{diff} \ x \ a))
\]

\[
diff \ () \ a = () \\
diff \ (a : x) \ a = x \\
diff \ (b : x) \ a = \text{diff} \ x \ a
\]

we replace MAP by the function "split"

\[
\text{perms } x = \text{split} \ 1 \ (\text{LENGTH} \ x) \\
\text{where} \\
\text{split} \ n \ n = f \ (x \ n), \\
\text{split} \ n \ m = \text{STRICT APPEND} \\
\quad \ (\text{split} \ n \ \text{mid}) \\
\quad \ (\text{split} \ (\text{mid}+1) \ m) \\
\text{where} \\
\text{mid} = (n+m)/2
\]

**Example 13**

The queens problem where the queens are to be placed on an \((n \times n)\) board in such a way that none checks any other. We use the solution of [4] where the board is represented by a list. The components of the list represent the columns of the board. Each component is an integer and its value represents the row of the board.
The algorithm starts from an initial position and either extends if the safe condition is satisfied or it modifies it to for a safe condition. It backtracks when a safe condition must be found by altering previously placed queens. Intuitively, we feel that fixing the initial positions and pursuing them in parallel to success or failure without backtracking will give us a parallel program. Allowing backtracking means that parallel paths may converge on the same route.

FOR 1 n initial
where
initial = soln (1,)
soln b = safe b -> full b -> b
FOR 1 n extend
where
extend q = soln (q : b)

By parallelising the list generator function FOR we easily obtain a parallel program. In fact we only transform the first occurrence of FOR otherwise the run time structure overwhelms the simulator.
To program the numerical method of solving Laplace's equation on a rectangular grid with given boundary values. This problem is programmed on the Data Flow computer [27] using a different approach to parallelism.

Initially the interior points of grid are given estimated (guessed) values and a new point on the grid is computed using the formula

\[ U_{ni} = \frac{(U_{ni-1j} + U_{ni+1j} + U_{nij-1} + U_{nij+1})}{4} \]

where \( n \) is the iteration step and \( i \) and \( j \) vary over the rows and columns respectively of the grid. The interior of the grid is iterated until successive values on each point differ by a given amount, which characterises the degree of accuracy of the approximation. The initial grid is given a constant value on all the interior points. The choice of initial value affects the number of iterations required to achieve convergence.
program

output
where
R = NO_OF_ROWS
C = NO_OF_COLS
BOUND_VALUE = ... || a R-list of C-lists

output = MAP grid (from 0)
grid 0 = INIT_GRID
grid n = FOR 1 R r
where
r i = FOR 1 C c

where
c j = BOUNDARY (i,j) -> BOUND_VALUE i j
(output (n-1) i (j-1) +
output (n-1) i (j+1) +
output (n-1) (i-1) j +
output (n-1) (i+1) j
)/4

BOUNDARY (i,j) = OR (i=1,i=R,j=1,j=C)
OR () = false
OR (a:x) = a # OR x
from n = n : from (n+1)

The function "from" it produces an infinite list which plays the role of the loop control variable in imperative programming.
The grid is represented as a list of lists. The output of the program is an infinite list denoted by the identifier "output". Each component of the list is a grid. Note the lazy evaluation mechanism of SASL enables output to be received from such an infinite computation. The pattern the computation follows is "compute a component of the list, print it and do the same for the rest of the list". When convergence is achieved we interrupt the computation. This can be determined by comparing the values printed out. A better solution where convergence is tested from within the program might be preferable but this program is adequate to demonstrate the idea of successive approximations being generated.

The algorithm adopted here can be thought of as a "bottom-up" method of solution. The evaluation of each new point comprises a rather trivial computation path. One way to extract parallelism is to divide the grid into sub-grids and compute each in parallel. The amount of parallelism obtained in this way depends on the size of the grid. But even a relatively small grid may involve a large number of iterations before convergence is achieved. This makes the amount of work on each sub-grid comparatively small.

In order to extract parallelism in the form where the whole evaluation process sub-divides into non-trivial smaller evaluations it seems we must adopt a "top-down" method of solution, where the result of the program is just the grid after a number of iterations. Intermediate grids
not being printed. In this way the evaluation of each point on this "final" grid involves a respectable amount of work.

program

output k = FOR 1 R r

where

r i = FOR 1 C c

where

c j = U k i j

U 0 i j = INIT_GRID

U n i j = BOUNDARY (i, j) -> BOUND_VALUE i j

(U (n-1) i (j-1) +
U (n-1) i (j+1) +
U (n-1) (i-1) j +
U (n-1) (i+1) j )/4

The + operators are marked as parallel +# and the arithmetic expression of the form A + B + C + D is rearranged to (A + B) + (C + D) in order to have balanced paths.

Since SASL does not support a package for simulating real numbers we were not able to test this program properly but only from the point of view of unfolding the recursions in parallel.
Example 15

To program a parser for Lambda Calculus strings defined by the following syntactic rules expressed in BNF (we use \( L \) instead of \( \) for typographical reasons).

\[
\text{wfe} = \text{var} \mid \text{lamb var} . \text{wfe} \mid ( \text{wfe} ) \mid \text{wfe wfe}
\]

\[
\text{var} = a \mid b \mid c \mid \ldots
\]

The syntax specification for a well formed formula of Lambda Calculus is that it is either a variable or a function or a bracketed well formed expression or a concatenation of well formed formulae. First of all immediate recursion is removed from the above syntax specification by introducing extra rules.

\[
\text{wfe} = \text{e1} \mid \text{fun}
\]

\[
\text{e1} = \text{e2} \{ \text{e2} \}
\]

\[
\text{e2} = \text{var} \mid (\text{wfe})
\]

\[
\text{fun} = \text{lamb var} . \text{wfe}
\]

\[
\text{var} = a \mid b \mid c \mid x \mid y \mid z
\]

where \( \{ \} \) indicate zero or more repetitions of the enclosed object. For simplicity the syntactic variable \( \text{var} \) is assumed to vary over just six names. For each syntactic variable a function is defined which recognises whether that entity occurs at the front of its input string. Such a recogniser function returns two results. A logical indicating success of failure to recognise the item and the remaining of the input string after the item has been taken
from the front of the input string. A recogniser corresponding to the left hand side of a syntactic rule uses recognisers corresponding to the right hand sides of rules.

A recogniser for a terminal symbol is a function which tests whether a particular string occurs at the front of its input string, ignoring leading spaces. So the test is string equality. In order to avoid defining a separate recogniser of each terminal, a function which takes a string as its input and returns a recogniser for that string is defined.

\[
\text{term pattern string} = f \ \text{pattern string}
\]

where
\[
\begin{align*}
f(p()) &= \text{false, string} \\
f()s &= \text{true, s} \\
f(p (%) :s) &= f(p)s \\
f(a:p) (a:s) &= f(p)s \\
f(p)s &= \text{false, string}
\end{align*}
\]

so the symbol \((\) is recognised by a function \(bra\) defined as

\[
bra = \text{term '(")
\]

The alternative \((|) BNF symbols are defined by a function \(bar\) which takes a list of alternative recognisers as its first parameter and an input string as its second parameter and tests whether the front of the string can be recognised by any of the recognisers.
bar () string = false,string
bar (rec1 : x) string = r1 -> true,s1
    bar x string
where
r1,s1 = rec1 string

thus var is defined as

var = bar (a, b, c, x, y, z)

and further for a , b , c , x , y , z , lamb

var = bar (map term ('a','b','c','x','y','z'))
lamb = term 'L''

similarly concatenation is defined

conc () string = true,string
conc (rec1 : x) = r1 -> r2 -> true,s2
    false,string
where
r2,s2 = conc x s1
false,string
where
r1,s1 = rec1 string

Using the function conc we can define a recogniser for
the category funct as follows

funct = conc (lamb , var , dot , wfe)
dot = term "."
Using bar and conc defining repetition is developed as follows.

```plaintext
repet obj = bar (obj.....obj , zero)
zero string = true,string
obj.....obj = conc (obj , repet obj)
```

thus finally in legal SASL

```plaintext
repet obj = bar (conc (obj , repet obj) , zero)
```

Now we are in a position to define the complete parser

```plaintext
wfe = bar (e1 , func)
e1 = conc (e2 , repet e2)
e2 = bar (var , conc (bra , wfe , ket))
```

To identify what can be done in parallel the functions bar and conc are analysed by unfolding their graphs. The graph of bar is shown in figure 5.6

```
figure 5.6 - program graph of "bar"
```

it suggests that the condition and right alternative of the conditional operator may be evaluated in parallel. In order to use the parallel operator \( \rightarrow \# \) already defined \( r1 \) is
replaced by \( \neg r_1 \) and the alternatives of the conditional are swapped.

The graph of \( \text{conc} \) is shown in figure 5.7

\[
\begin{align*}
\text{conc (rec1 : x) string} \\
\quad &\quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \qua
\texttt{ONEOF l r = \textasciitilde\textbf{hd} l \rightarrow \# r \ ; \ l}
CHAPTER SIX

Results

In this chapter we analyse the results, obtained by running the programs developed in chapter five, on the parallel evaluator (see chapter three and four) and comment on the method by which parallel programs are derived from sequential ones.

Simulation results are presented in the appendix, in the form of tables. Tables numbered n.1 and n.2 correspond to the example program numbered n in chapter five. What do the tables mean?

As we have discovered in chapter five a (parallel) program task decomposes into a tree of sub-tasks. A special case of this is the program which tests a directed graph for the bipartite property (Example 5) where its evaluation only occasionally decomposes into two sub-tasks and the rest of the time it consists of a single task.

The evaluator has a choice of evaluation schemes at its disposal. This is controlled by an input parameter (see below about "strategy") of the simulation. The most obvious schemes are two, the totally sequential scheme where no parallelism is invoked at all and the other is the maximally parallel (most eager evaluation) scheme where as soon as a sub-task is created it is assigned to an evaluator. The sub-task is processed independently of the
main task and its associated evaluator is an assistant to the one processing the main task. In between these extreme evaluation schemes there exist a number of evaluation schemes each dictating when evaluations are "forked out" from ongoing evaluations. Thus certain tasks which would be processed in parallel under the maximally parallel scheme are evaluated in sequential order under an "in-between" scheme. Note that although the evaluator's behaviour seems to vary between eager and lazy evaluation this is not strictly accurate since call-by-parallel has replaced call-by-need (see chapter two) even in the absence of parallelism due to the dictates of a particular scheme. In this case the evaluator simulates the parallel evaluation of sub-tasks. Each evaluation scheme is called a "strategy" of the evaluation mechanism. Below we explain how strategy is quantified.

The simulation we have constructed sets out to discover how to exploit the parallelism "inherent" in the programs of chapter five by testing different parallel evaluation schemes (strategies). The effect of each scheme is measured by the resulting length of computation (number of main evaluator's steps).

The performance under each parallel evaluation scheme is calculated as a percentage improvement over the length of computation under the totally sequential scheme. Thus in table 1.2 for instance a particular strategy (horizontal axis) of 10% (see below) achieves 60% gain in performance
A particular strategy dictates when each evaluator is "off-load" (logically) a sub-task to an assistant evaluator. This is when a parallel activity is to be set up. A strategy models the degree of parallelism employed in a real machine consisting of multi-processors for a given program.

So under a strategy a certain amount of work is shared amongst evaluators and a corresponding improvement over the sequential strategy (just a single task, the main one) is expected.

A strategy amounts to an assumption concerning the pattern of resource allocation in a real machine. In this study we have assumed that an evaluator gets the benefit of an assistant after it has performed a certain amount of work. During this time it may have generated some or no new (sub-)tasks. In the former case these are assigned to assistant evaluators as the particular strategy dictates.

The condition of unbounded parallelism (number of assistants or number of "off-loadings") is assumed.

The "amount of work" (or "time") referred to previously is based on three types of measurement

(a) the number of steps

(b) the number of COLON steps (list cell creations)
(c) the number of APPLY steps (function applications)

we have found that all three methods of measuring the amount of work give approximately the same results.

Each strategy is represented by a percentage, input to the simulator. For example, a strategy of 10% indicates that each evaluator is allowed to obtain an assistant whenever the work it has done exceeds 10% of the total amount of work the program would have entailed under the totally sequential scheme. This is done in order to meaningfully compare results from programs of different computation lengths (number of steps). If a program is evaluated under a more eager strategy, say 5%, modelling the case where the machine is bigger, we wish to discover the corresponding effect on the performance of the program. Thus 0% represents the maximally parallel scheme and 100% represents the totally sequential scheme.

The simulation has a twofold significance. On one hand we use it to discover the amount of parallelism in programs and on the other it indicates a scheme of machine program organisation suitable for an environment which incorporates parallelism.

The histograms, tables numbered n.1, give us an idea of the run-time profile of each program under the 10% strategy. The vertical axes of tables n.1 show the number of evaluators processing tasks and the horizontal axes show time in terms of computation length. We discovered that
the shape of histograms generally remains the same for different strategies so only the 10% case is shown. A histogram indicates the amount of work that can be done in parallel over time. We also compare histograms against our intuition about what programs do.

Parallel programs

In this section we comment on what we have discovered about the method of deriving (by hand) parallel programs from initially sequential ones. First, we have found out that parallelism needs to be expressed (effected) by two language constructs which are introduced into SASn for the purpose of expressing parallel programs. These are the annotation symbol "#" which modifies a primitive operator to a parallel primitive operator. In particular we note that the parallel non-strict primitive operators \&\# (PAR-AND), |\# (PAR-OR) and ->\# (PAR-CONDITIONAL) express the notion of speculative parallelism where the evaluation of one of their operands is initiated in anticipation that its value might be needed and terminated forcibly when otherwise.

The other parallel construct we found to be needed is the call-by-parallel parameter passing mechanism expressed (forced) by the system function STRICT which operates on a function and its two parameters. It "passes" evaluated the two parameters to the function. Any other more complex case of call-by-parallel was handled by defining an
appropriate (user) function in terms of STRICT and auxiliary functions (see chapter two).

Note that our approach does not rely on explicitly creating and synchronising "processes" so that we avoid the problem of the run-time management of parallelism at this level. We have resorted to the use of parallel constructs, taking caution against non-termination, for the purpose of experimentation, of controlling the "grain" of parallelism, avoiding non-useful parallelism and finally since call-by-need cannot be replaced by call-by-parallel (a case of call-by-value) without introducing non-termination (see chapter two).

In order to identify parallelism in a program we proceed from the top (outer) level function definitions to the inner ones. Each time we enter a level the "grain" of parallelism becomes finer. The corresponding program graph identifies the data dependencies. For instance in the example 15 (parsing strings), the graph of the function "conc" indicates a sequential function whereas parallelism was identified at the level of the function "bar" which has a similar structure as "conc" but no prohibiting data dependencies.

Parallelism can be seen from performance graphs to be most enhanced if the program graph is balanced in the sense that the sub-expressions to be evaluated in parallel are of similar complexity. This is enforced when the Divide-and-
In some cases parallelism manifests itself as parallel evaluation of a list's components (when its length is finite). For example expressions of the form

```
a : b : c .... : ()
```

![Figure 6.1 - Program graph of a list evaluation](image)

whose graph, shown in figure 6.1, suggests that the representation of a list by a two field data structure (a cell) gives a rather unbalanced tree of tasks.

![Figure 6.2 - Transformed program graph](image)

In order to obtain a balanced tree the operator : is replaced by the parallel function APPEND (defined in chapter five), the graph of the transformed expression is shown in figure 6.2. The functionality of the function APPEND requires us to change the components a, b ... into
1-lists (a,), (b,), ..., this is a rather ad-hoc solution. Keller [41] avoids the overhead introduced by the function APPEND by proposing a different data structure to the list cell.

Here a connection with the work of Darlington [45] is apparent. A system of formal derivation of parallel programs from initial sequential ones or from initial specifications of programs is desirable. For example it is interesting to speculate whether the parallel matrix multiplication program (Example 9, chapter five) could be formally derived from an initially sequential one.

The parallel program for the queens (Example 13, chapter four) was obtained by reprogramming where backtracking was eliminated in favour of forward moves. Here we also note a certain inelegance since a path of forward moves which fails to arrive at a solution is represented by the empty list "()" which appears in the output of the program since it is generated. The introduction into SASL of set expressions [35] which evaluate to lists avoids the generation of unwanted components of the output list.

The case of the numerical program for solving a partial differential equation on a rectangular grid (Example 14, chapter five) required reprogramming in order to compute the result of the computation in a "top-down" fashion instead of the "bottom-up" method of the initial sequential
The run-time results

In this section we analyse the results, shown in the appendix, of running the example programs, developed in chapter five, on the simulator we have constructed (see chapter three and four).

The tables 1.2, 2.2, 3.2, 5.2, 9.2, 10.2, 15.2 indicate that the performance of the example programs 1, 2, 3, 5, 9, 10, 15 is related linearly to the strategies of parallelism. Tables 4.2, 7.2, 8.2, 12.2, 13.2, 14.2 show a kind of exponential relationship. This must be due the fact their tree of sub-tasks are well balanced.

The example 6 (computing a zero of a polynomial) was not tested due to lack of real numbers in SASL, though we could have worked with some scaling. Example 11 (generating the partitions of an integer) turns out to be essentially sequential.

The histograms, tables 1.1 - 15.1 give us the profile of the parallel evaluations over time. These agree with our intuitive understanding of what programs do. For example table 5.1 (testing for the bipartite property on an undirected connected graph) where its evaluation can at most decompose into two parallel (sub-) evaluations. The histograms 2.1 (testing a directed graph for a cycle) and 3.1 (computing a vertex where a cycle starts) have a steep end since the completion of some sub-task causes the termination of all other tasks. The histogram of example 15
(parsing strings), table 15.1, indicates that for most of the time there is a single task (sequential evaluation for most of the time) since the sub-tasks terminate rather quickly. This is due to the fact that sub-tasks test the legality of a sub-string of the input string to the parser. Also parallelism is limited since it is only identified with one function, namely "bar", where all the other functions at the same level (grain) as "bar" are sequential. Table 9.1, corresponding to the Sort-by-merge program, indicates that for a large part at the end of the computation there is a single task (the main one) due to the fact that finally two large lists have to be merged to give the result sorted list of numbers. Merging is a sequential "operation". Valleys in the histograms indicate periods of sequential operations. Example 10 (computing a relation in a library) showed a large number of lock steps (see chapter four) due to the fact that all parallel evaluations search two global association lists.

The more or less symmetrical histograms indicate that at the beginning and at the end of computation the number of sub-tasks is low, exponentially increasing (decreasing) in between. This is due to the fact we have a binary tree of sub-tasks and the balancing tends to be good in such case.

Example 11, generating the partitions of an integer, exhibits an interesting point. Under the most eager strategy (0%) it yielded 65% gain over the length of the
totally sequential evaluation whereas the application of memo-isation [36] yielded 67%. This strongly suggests that performance gains are obtainable by means other than that of parallelism.

Finally we observe that the maximum number of parallel activities is 18 and although we have experimented with "toy" programs we can speculate that there will be maximum demand upon the resources of a real machine only for a rather limited period.

In table 16 the SPEED UP FACTOR is shown for the maximally parallel strategy (0%). This is related to the PERFORMANCE gain shown in the vertical axes of tables 1.2 - 15.2 calculated as

\[ \frac{100}{100 - \text{PERFORMANCE}} \]
CHAPTER SEVEN
Conclusions

The work presented in the previous chapters has focused on two issues. The nature of a parallel implementation of SASL and the amount of parallelism in particular programs exploited by the implementation.

The implementation is based on the SECD implementation of SASL. This has been extended with primitives which handle the interaction of a whole regime of SECD machines, referred to as evaluators. The evaluators combine their effort in processing a single program task. This is possible because a program task decomposes to sub-tasks where each of those may decompose further and so on. A program which simulates a regime of evaluators, an unbounded number of which is assumed, has been constructed.

The evaluation of a program gives rise to a spectrum of behaviours in the simulator each determined by a strategy of spawning. Each strategy of spawning represents a particular degree of parallelism employed during the evaluation of a program. The spectrum varies between a totally sequential computation where just a single evaluator is employed to process a program and a maximally parallel computation where a new evaluator is employed whenever a computation splits into sub-computations. Each behaviour between these extreme cases is characterised by the fact that a new evaluator is employed under a certain
constraint. This is imposed by having each evaluator working on some task obtain assistant evaluators after it has performed a certain amount of work and providing it has generated sub-tasks.

The parallelism of a program is investigated by evaluating it under different strategies and noting the corresponding performances. The performance measure is based on the number of evaluator's steps it takes to process a program to completion. Each step is equivalent to the "execution" of an SECD machine instruction. We expect results obtained to hold true for other types of implementations where we have different steps, for example SK machine steps.

We discover by associating a strategy with the degree of parallelism employed in a multi-processor machine, that there is often a linear relationship between the performance of programs investigated and the degree of parallelism. This suggests that in a realistic situation, providing the parallel implementation can be efficiently supported doubling the size of the multi-processor machine approximately halves the run-time.

The parallelism of programs manifests as simultaneous evaluation of the operands of primitive operators and as simultaneous call-by-value on the parameters of functions. The parallel conditional operator gives rise to the notion of an "irrelevant" evaluation where its condition and
alternative(s) are evaluated in parallel. The evaluation of the alternative(s) is initiated in anticipation its value might be needed. If it turns out that it is not needed then it must be identified and terminated. Thus in order to extract parallelism from programs the lazy evaluator must be forced to do some work.

There are two problems with replacing call-by-need by call-by-value. On one hand it may introduce non-termination and on the other not all function calls offer the opportunity for useful work to be done in evaluating the actual parameters in parallel. The latter is also true with instances of primitive operators where the evaluations of their operands are ordered by a data dependency or one of them is rather trivial. In both cases parallelism cannot be introduced usefully. The approach we take is to introduce source annotations which mark the primitive operators which are to be interpreted as being parallel. Call-by-value is expressed in terms of primitive operators. We envisage that further work might be able to identify such operators partly automatically but this is, in general not computable. The annotations direct the compiler to produce parallel code (parallel instructions corresponding to parallel operators) which causes the evaluator to generate a tree of tasks. A similar use of functions like "STRICT" directs the simulator to evaluate certain arguments of functions in parallel (call-by-parallel). A task is associated with the operand of a parallel operator. The
strategy of spawning, mentioned above, causes assistant evaluators to "take away" tasks, so that when the evaluator comes to process them they are evaluated already.

In order to identify in a program the parallel operators, and simultaneous call-by-values a program is represented as a graph of data dependencies. The definitions of names are used to unfold the graph discovering the data dependencies. In several cases in order to obtain a balanced tree of tasks the original program is transformed to a better parallel program by applying a programming technique known as divide and conquer.

We have gained considerable experience with the parallelism of a variety of programs. The structure of a parallel program seems to be of the following three forms.

1. The Divide and Conquer form where a program's evaluation recursively sub-divides into evaluations of similar complexity, the program for matrix multiplication, developed in chapter four, is an example of a program possessing this form.

2. The speculative form where evaluations are initiated in anticipation their result (value) will be needed. The parser for Lambda Calculus expressions is an example of this form.

3. The evaluation of a program occasionally requires the parallel evaluation of certain sub-expressions before
it continues sequentially, for example the program which tests whether an undirected connected graph is bipartite.

Note that all cases of parallelism concern deterministic programs. The case where parallelism is introduced by non-deterministic constructs has not been dealt with. The introduction of non-determinism enables a certain class of programs to be programmed in (near) applicative style [42]. This notion of parallelism is beyond the scope of the present study.

Parallelism may be extracted from non-numerical as well as numerical problems alike. The example programs investigated cover a wide range of applications.

The final word of course lies with the computer architects. What we have examined here is the "logical" aspects of parallelism, what can be done in parallel and for what programs. There have been a number of proposals for multi-processor machine designs [31,39,40] which set out to support efficiently a notion of parallelism rather similar to the one investigated in the present study. The results of the present research have important implications for such research. It is clear that many algorithms which would not take advantage of such hardware can be transformed into more appropriate forms. It may be that appropriate language constructs would lead to the natural production of parallel programs.

It seems that a risky philosophy of task initiation is
almost essential if advantage is to be taken of inherent parallelism and consideration should be given to the efficiency of the killing process for irrelevant computations.
REFERENCES

1. J. Backus, Can programming be liberated from the Von Neumann style? a functional programming style and its algebra of programs CACM 21, 8 1978, pp 613-641


3. P.H. Welch, Lambda Calculus lecture notes 1977 University of Kent Computing Laboratory


5. R. Morrison, S-ALGOL language manual, CS/79/1 Depart. of Computational Science University of St. Andrews


10. Wozencraft & Evans, Notes on Programming Linguistics, MIT 1979

11. D.A. Turner, SASL language manual, CS/75/1 Depart. of Computational Science University of St. Andrews


14. M. O'Donnell, Computing in systems described by equations, Lecture Notes in Computer Science vol. 58

15. P. Landin, The mechanical evaluation of expressions, Computer Journal vol. 6, 1964, pp 308-320


17. J. McCarthy et al., LISP 1.5 programmers manual, MIT Press 1965

18. P.H. Welch, Some notes on the Martin-lof/Tait proof of the Church-Rosser theorem, University of Kent Computing Lab. 1975


23. A. Mycroft, The theory and practice of transforming call-by-need into call-by-value, CSR-88-81, Depart. of Computer Science University of Edinburgh


26. D.A. Turner, Another algorithm for bracket abstraction
    Journal of Symbolic Logic July 1979

27. P. Wegner, Programming languages Information
    structures and machine organisation
    McGraw Hill 1968

28. W.H. Burge, Recursive Programming
    Addison Wesley 1975

29. D.A. Turner, An implementation of SASL TR/75/4
    Dept. of Computational Science
    University of St. Andrews

30. J. Stoy, Denotational semantics
    MIT Press 1980

31. R. Keller et al.,
    A loosely-coupled applicative
    multi-processor system AFIPS 79 pp 613-622

32. H. Baker & C. Hewitt,
    The incremental garbage collection
    processes SIGPLAN Notices (ACM) vol. 12,
    no. 8, 1977

33. E.W. Dijkstra,
    Cooperating sequential processes
    Programming Languages pp 43-112
    Genuys (Ed.) Academic Press 1968

34. W. Burton & R. Sleep,
    Executing a virtual tree of processors
    School of Computing Studies,
    University of East Anglia, Norwich

35. J. Darlington,
    A synthesis of several sorting algorithms
    Acta Informatica vol. 11, 1978, pp 1-30

36. D.A. Turner, The semantic elegance of applicative
    languages Proc. MIT/ACM Conf. on Functional
    programming languages and
    computer architecture
    Portsmouth, New Hampshire, October 1981

37. P. Treleaven,
    Principle components of a Data Flow computer
    SIRM/216 University of Newcastle upon Tyne
    Computing Laboratory 1979

38. D. Michie, Memo-Functions
    Experimental Programs 1966-67
    Dept. of Artificial Intelligence
    University of Edinburgh
39. R. Sleep & W. Burton,  
Towards a zero assignment parallel processor  
Proc. 2nd. int. Conf 1981 on distributed  
systems

40. J. Darlington & M. Reeve,  
ALICE a multi-processor machine for the  
parallel evaluation of applicative programs  
programming and computer architecture.  
Portsmouth, New Hampshire, October 1981

41. R. Keller,  
Some theoretical aspects of applicative  
multi-processing  
Lecture Notes in Computer Science vol. 88,  
pp 58-74

42. P. Henderson,  
A purely functional operating system  
Functional programming and its applications  
J.Darlington, P.Henderson, D.Turner (eds)  
C.U.P 1982

43. J.R. Hindlay et al.,  
Introduction to Combinatory Logic  
C.U.P 1972

44. J. Reynolds, Definitional Interpreters for higher order  
programming languages  

45. P. Treleaven et al.,  
Combining control flow and data flow  
APPENDIX I
In a directed graph to compute the reachability relation.
TABLE 2.1

TABLE 2.2
TABLE 3.1

Program to compute a cyclic vertex of a directed graph.

TABLE 3.2
EVALUATORS

PERFORMANCE (%)

TIME (x100 M/C STEPS)

STRAIGHT

Evaluators

Graph is incorrect.

To test whether an undirected connected

TABLE S.2

TABLE S.1
TABLE 8.1

Program 8x8 matrix multiplication
TABLE 8.2

[Diagram showing a scatter plot with labeled axes: PERFORMANCE (%) on the y-axis and STRATEGY on the x-axis. There are a few data points marked with ▼.]
TABLE 9.2

TABLE 9.1
TABLE 12.1

To compute the permutations of the numbers 1, 2, 3, 4, 5
TABLE 13.2

En ekg board To solve the greens problem on

TABLE 13.1
TABLE 14.1

Program to numerically solve Laplace's equation by the grid method

TABLE 14.2
<table>
<thead>
<tr>
<th>PROGRAM</th>
<th>SPEED UP FACTOR</th>
</tr>
</thead>
<tbody>
<tr>
<td>the reach of a graph</td>
<td>6</td>
</tr>
<tr>
<td>cyclic graph</td>
<td>14</td>
</tr>
<tr>
<td>start of a cycle</td>
<td>10</td>
</tr>
<tr>
<td>terminal vertices</td>
<td>5</td>
</tr>
<tr>
<td>bipartite graph</td>
<td>3/2</td>
</tr>
<tr>
<td>hanoi</td>
<td>10</td>
</tr>
<tr>
<td>matrix product</td>
<td>5/2</td>
</tr>
<tr>
<td>merge-sort</td>
<td>3</td>
</tr>
<tr>
<td>library relation</td>
<td>6</td>
</tr>
<tr>
<td>permutations</td>
<td>5</td>
</tr>
<tr>
<td>6 queens</td>
<td>5</td>
</tr>
<tr>
<td>laplace grid</td>
<td>3</td>
</tr>
<tr>
<td>parser</td>
<td>8</td>
</tr>
</tbody>
</table>

The speed up factor is related to the PERFORMANCE axis in tables 1.2 - 15.2 by the formula:

$$\frac{100}{100 - \text{PERFORMANCE}}$$
Just return binding if done, terminate computation

prefix operators

abstract a file, calling the file, creating saved on computer first file

structure a pre-end (structure)
```
function compose(f, g) {
  return function(x) {
    return g(f(x));
  }
}
```

```
function identity(x) {
  return x;
}
```

```
function transformer(name) {
  return function(x) {
    return function(func, name, target) {
      return func(target, name);
    };
  }
}
```

```
function strict(name) {
  return function(func, name, target) {
    return func(target, name);
  };
}
```

```
function number() {
  return function(func, name, target) {
    return func(target, name);
  };
}
```

```
function logcat() {
  return function(func, name, target) {
    return func(target, name);
  };
}
```
( )
forward 
print
forward 
cons
forward 
append
forward 
append
forward procedures

and

append a

long for recursion
the any
the and
the any
define a suspended mapping nil false the any nil

( )

prefix apply a
prefix apply
prefix checklist

prefix colon

let this = cond

prefix end

let g = length

let f = length

let a = length

let b = length

begin (II)

let append = nil

else (II)

append (II)

append 5 = f then f =

append a = internal function for appending lists (II)
begin

end

begin

end

begin

end

begin

end

begin

end

begin

end

begin

end

begin

end

begin

end

begin

end

begin

end

begin

end

end
let leftharm =condepx
let rightharm =condepx
let shmp =condex

let leftharm =condepx
let rightharm =condepx
let shmp =condex
true false false

begin end
begin end

procedure condex(out) return

procedure condex(out) return

procedure condex(out) return

procedure condex(out) return

end
begin
begin

begin
begin

begin

begin

begin

begin

begin

begin

begin

end
begin
procedure condex(out) return

end
procedure condex(out) return

end
procedure condex(out) return

begin
begin

begin

begin

begin

begin

begin

begin

begin

end
procedure condex(out) return

end
procedure condex(out) return

end
procedure condex(out) return

end
procedure condex(out) return

begin
begin

begin

begin

begin

begin

begin

begin

begin

end
procedure condex(out) return

end
procedure condex(out) return

end
procedure condex(out) return

end
procedure condex(out) return

}
else
{
}
if have("/") then infix(DIV, v, opexp(5)) else oldv
while v = oldv
if prio < 4 do
repeat
< oldv := v
v := if have("+") then
  if symb="#" then nextsymb; p(infix(PLUS, v, opexp(4)) )
  else infix(PLUS, v, opexp(4))
else oldv
while v = oldv
if prio < 3 do
repeat
< oldv := v
v := if have("=") then
  if symb="#" then nextsymb; p(infix(EQ, v, opexp(3)) )
  else infix(EQ, v, opexp(3))
else if have("=") then infix(NE, v, opexp(3))
else if have(">") then infix(GT, v, opexp(3))
else if have("<") then infix(LT, v, opexp(3))
else if have("<=") then infix(LE, v, opexp(3))
else if have(">=") then infix(GE, v, opexp(3))
else oldv
while v = oldv
if prio < 2 do
while have("/") do
  v = infix(PARAND, v, opexp(2))
if prio = 0 do
procedure name (expression) {
begin

If symbol = NAME then
  next symbol

if symbol = CONSTAT then
  print symbols

if symbol = APPPLY then
  print symbols

}
begin
let ds = clause
while have "," or starter do
begin
    let d = clause
    if d.defn.name = ds.defn.name then
        if d.its.defn is map and ds.its.defn is map then
            construct a list of alternatives from the two clauses
            ds.its.defn = trys( cons( ds.its.defn ),
                                   cons( d.its.defn, nil ) ),
        else
            if d.its.defn is map and ds.its.defn is trys then
                begin
                    add clause d to list of alternatives for ds
                    list = ds.its.defn, clauses
                    while list(t1) = nil do
                        list = list(t1)
                        list(t1) = cons( d.its.defn, nil )
                end
            else
                error "Inconsistent definition of " ++ d.defn.name, the.id )
        else begin
            distinct names
            d.next.defn = ds
            ds = d
        end
    end
end

procedure clause( -> ptr )
begin
let names = namelist
if have"=" then defn( names, expr, nil, nil )
if sym = NAME then
  let f = FAIL
  and
  end

= 0

while p = null and (and (name = p.0) do p = p.1

begin
  procedure member (put name \ remembering | do | fail

begin
  procedure formal + put

begin
  else map \ f, the

end

end

end

end

end

end

end

end

end

end
procedure nilbag ( )
begin
end

if have( ) 
then cons( ) 
else if structure

then mustuse( )
begin
if have( ) 
then
end
else
mustread( )
begin
if have( ) 
then
end
else
mustread( )
end

if this.value.ts num then this.value.the.num( ) = value.the.num( ).
begin
if have( ) 
then
end
else
mustjump( )
begin
if jump = nilbag
then
else
mustread( )
begin
if member( ) 
then
end
begin
if unrecovored do
   procedure syntax (string target)
   begin
      if have (target do unrecovored) then false = false.
      if have (target = unrecovored) then 1 = 1.
      while symb = target and symb = "" and symb = "" do next symb
      procedure mustbreak (string target)
      end
   procedure have (string target = do obj)
      symb = "" of symb = ""
      where or symb = ""
      or symb = ""
      or symb = ""
      or symb = ""
      or symb = ""
      procedure terminator (do obj)
      end
   procedure separator (do obj)
      end
   else n = formal
   else cons (n, null)
      if terminator then cons (n, null)
      if have ("" = then
      if n = formal
   end

begin
end
errorflag := true
unrecovered := true
message given := true
write """"nSyntax: """" target: """" expected where """" symb, """" found in: 'n'n"

end

procedure show text
begin
let p := buffer ptr
let lines := 2
' find start of last two lines in the circular text buffer
repeat
   p := if p = 0 then buffer size else p - 1
   lines := if text buffer( p ) = '"'n*'"' than lines - 1 else
   if text buffer( p ) = '"' or p = buffer ptr then 0
       else lines
while lines > 0
   ' write out those lines
   repeat
      p := ( p + 1 ) rem buffer size
      write text buffer( p )
   while p ^= buffer ptr
   message given := false
end

lexical analysis routines ( procedure identifier above )

procedure get char
begin
ch := if eof( input file ) then ENDCH
else read( input file )
if echo and input file ^= s.1 do write ch
buffer_ptr := buffer_ptr + 1 \rem buffer.size

if ch = \"\"n\"\" then \( ps \div 8 + 1 ) \times 8

else ps := 1

if message.given and (ch = \"\"n\"\" or ch = ENDCH) do show text

end

procedure layout

begin

while ch = \"\"n\"\" or ch = \"\"t\"\" do getchar

pos := ps

end

procedure nextsymb

begin

procedure read_word( string first -> string )

begin

let name := first

getchar

while letter(ch) or digit(ch) or ch = \"\"_\"\" do

{ name := name ++ ch; getchar }

name

end

procedure try( string s )

begin

symb := ch
if ch = "QUOTE then
   if ch = "QUOTE then
      if ch then
         if ch then
            print
         getcharacter
      else else
         print
      else
         print
      end
   else
      print
   end
end

begin getcharacter
   let c = chr ch
   getcharacter
begin getcharacter
   end

begin
   return = "",
   sym = CONSTANT
   begin
      (c) = sym
   end
   nextchar
begin
   while ch = "-" and ch = "_" and ch = "=" and end do getcharacter
begin
   if sym = "(" then
      getcharacter
      sym = ""
   end
end

begin
   if sym = "" then
      print
   getcharacter
end
```
procedure level (count, errorflag) do

default

begin

false

begin

true

end end

end.
```

procedure pushstack (Input Team / No stack limit assumed)
end

result := result "no output" do result (child) := death;
if stack = null do write "stack IS NULL."
stack := stack' (next)
let result := stack' (top)
end

procedure popstack (-put)

if value = false
procedure outcycle

if value = true
procedure incycle

structure dump (counter Coded/Carrying Sub. Cycle / Emt Eny. Dump)
structure main
structure print

default
true
false
begin

procedure cont (cont : code ; c : cycle)
end

begin

DUMP := DUMP (DUMP)
ENV := DUMP (ENV)
END CYCL := DUMP (END CYCL)
CODE := next C

if ENV = null then
    raise "null ENV"
else
    ENV := ENV (ENV)
end

if SUB CYCL = null then
    raise "null SUB CYCL"
else
    SUB CYCL := SUB CYCL (SUB CYCL)
end

if CODE = null then
    raise "null CODE"
else
    CODE := CODE (CODE)
end

begin

procedure load cont
end

procedure trace pa

DUMP := DUMP (DUMP)
ENV := DUMP (ENV)

procedure save cont (cont : code ; c : cycle)
end

begin

procedure cont (cont : code ; c : cycle)
end

end

begin

SUB CYCLE, RES SLOT

Assume CODE, ENV, DUMP STACK

if "father"(the set) do lockup("father")
if "father"(the set) = lockup("father")
else
else
else

procedure lockup(count p s)

if ps(lize) = samp do act regist
else if is top

let father = ps then stripup("father")
else if father = ps then stripup("father")

end

SUB CYCLE = sub cycle
CODE = code
unlock = false
begin
  "name"
  case SIGCYCL of
    CODE 1: cond:
      and
      if val is suspended then
        begin
          val = lockup (CODE, ENV)
          CODE is id.
        end and cond
        pushstack (CODE, the cond)
        begin
          CODE is cond:
            case true of
              behaviour of a computation
                measure it in terms of APLY/M/C-instructions
                register the size of the computation
          end case
        end begin
  end begin

end begin

procedure eval cond
E-4102 Slatten page 24
case null cycle of
CODE IS (partial)

else CODE (right fork)

cont act if a = true then CODE (left fork)

else cont

pushstack(CODE (right fork, CMD, a))

if STACK top last logic then

cont act CODE (left) ELSE CODE (right)

save control (none)
arith block
CODE = prefix

Infix block
CODE = infix

Prefix block
CODE = prefix

end load cont
pushstack (current CODE, ENV)
BEGIN
if map on CODE is true
begin

default: write "never"

PS (this sett) = false; load cont; else monitor
PS (this sett) = false; pushstack (ERR); (COND, b)
default: if b is real then
begin
PS (this sett) = false; pushstack (ERR); (COND, a) name"

false:
PS (this sett) = false;
al; else a = pushstack (null) pushstack (a); load cont
true:
case a of
(1) = a = stack (top) ; let b = stack (post); top

null
DIN

let code = infix( APPLY ,
    infix( APPLY , compose ,
        CODE( e1 ) ) ,
    CODE( e2 ) )
cont.at( code , "none" ) ) , INCYCL )

PLUSPLUS

let code = infix( APPLY ,
    infix( APPLY , append ,
        CODE( e1 ) ) ,
    CODE( e2 ) )
cont.at( code , "none" ) , INCYCL )

EQ,NE equal.block
PAROR,PARAND or block
default 1 CODE is arith operation or relation

case SUB.CYCL of
"none"
begin
save cont( "once" )
cont.at( CODE( e2 ) , "none" )
end

"once"
if STACK: top is suspended then
begin
begin

if Stack() rest = top is suspended then

end else

INC

if Stack() top is suspended then
default : "machine"

else

conf at Code at 1 , "none"

save cont ( "machine"

pushstack (sofar)

begin

next loop

pushstack (top)

begin

let = Code (Infix op)

let sofar num then

let sofar = pushstack

begin

end else

INC

conf at cont ( right op ) , "cannot"

save cont , "none"

let right op = pushstack
end of default if not found
and of arithmetic operation

push stack result
default null; never occurs

(let q = (e) if
(let q > (e) if
(let q = (e) if
(let q < (e) if

REM number = 0
DIV number = 0
TIMES number = 0
MINUS number = 0
PLUS number = 0

if result = case (of) push stack

let q = VAL1 the num

begin

end else
push stack (or) load, cont

let er = ERR CODE (number)

val2 = VAL2

if val1, num or VAL2 first num then
begin

end else

INVERT

cont and (or) case (of) "case" push stack (or)
pushback(ENV)
pushback(formal the const)

let max = int(x, ENV, formal the const)

begin
if formal is const then
else
load const
pushback(ENV) ; needed
end

if formal is id then begin
ENV = defn(formal, arg, ENV)

let formal = popstack
let arg = popstack

begin

procedure block

name once block

name once block

case and cycle of

procedure apply block


ink (previous arg, at ) \ target arg

else cons prevous arg

if previous args = nil then cons \ target arg nil

procedure link input previous args \ target arg - output

\-0.3. System page 43
save_cont( "from.equal" )
cont.at( next, "twice" )

if_formal_is_repetition then
begin
  let itsval = lookup( formal( the.rpt ), ENV )
  let next = infix( EQ, formal( the.rpt ), arg )
pushstack( itsval )
pushstack( arg )
save_cont( "from.equal" )
cont.at( next, "twice" )

end else

if_arg_is_suspended then
begin
  pushstack( formal )
save_cont( "binding" )
cont.at( coarse? arg ), "none" )

end else

if_arg_is_cons then ! formal is cons too
begin
  pushstack( formal( tl ) )
pushstack( arg ( tl ) )
pushstack( formal( hd ) )
pushstack( arg ( hd ) )

  save_cont( "frombinding" )
cont.at( CODE, SUB.CYCL ) ) INCYCL

end else
begin
  pushstack( FAIL )
load_cont
CODE( z )
begin
  let r = arrz, Apply, rator,
  let rator = postack
begin
  if any = FAIL and CODE( prefix ) = APPLY then
    begin
      begin
        binding, done
      end
      end
      ENW = any
begin
  end while
  load cont to the "binding-counter"
push(rator)( FAIL )
the rest of the binding
throw : postack
throw : postack
let any = FAIL then begin
let any = postack
begin
may think of FROM a sub-part of binding
know there is more, no need to use the DUMP
set ENW to any, and go directly to b stage
return FAIL to FROM-counter or to-dump or
"from-binding" sure there is more binding to do on the STACK
begin
and
Let result off ICC = popstack
begin
    from ccmt.
    select block
    and
    load cont
    pushstack (m
    )
    let l = basic (rator, rand)
    begin
        if rator is empty then
            cont = (CODE, \"select\")
            pushstack (rand)
            pushstack (rator)
        begin
            else
                load cont
                pushstack (rator, apply, rator, rand)
        end
    end
    INC
    INC
pushstack( traceback )
begin
  let traceback = popstack
  let anglist = popstack

  begin
    case SUB_CK of 
    procedure try block
      begin
        end
      end
    INCOL
      cont at CODE , select
      pushstack( index )
      pushstack( unexpanded )
    begin
      else
        load cont
      end
          pushstack( array , APPLY , unexpanded , index )
      if unexpanded last cont then 
        let index = popstack
        let unexpanded = popstack
        must be 1 at
        begin
          default
          from end
          end
          load cont
          else pushstack( FAIL )
        let Result of EG = TRUE then pushstack( ETV )
  end
end

let result of try = popstack

begin
  if try is still in try then
    let result of try = popstack

end

end

end
def pushstack(flag, to, done)

let flag to done = still in try (cons and any)

begin
  while iteltoverses nil do
    let iteltovers = restargs
    begin
      end else
        load cont
        pushstack(cons clause body, any)

\end{verbatim}
CON CERI NUCLE P "none"

let newv = (v) { let newv = (v) and
let newv = (v) + the closure
begin ox or not or ox
load cont
stashpack arglist
let arglist = stashpack
ашe arglist, classess, f1)
( )
pushstack arglist
pushstack classess, 4t)
pushstack arglist, for new thr
begin
if classes = nil then
if classes = nil then | new thr
end else
load cont
pushstack newv
let newv = (v) ge on, arglist
let ge on = consl dr, classess
let ok dr is (class class)
let thr new = classess | the closure
begin
if it is another thr then
let arglist = classess | for new thr
begin

and

cont & ( CODE & 1 = "none"

save cont & "none"

begin

"none"

and

case SUB_CYCLE of

| a recursive instruction of the m/c

procedure equal block

end

end

load cont

{ pushstack + arry APPLY 1 Byte index }

end else if 1 = 0

else pushstack + 0

load cont

{ count all characters of "coarse" }

if 0 & 1 is suspended then

let ref = first and

design

{ index the num | 1 then

and else if 1 & 1 = 1

end

INCICAL

G-3490 System page 44
if VAL1 is num and VAL2 is num then

begin

case VAL1 of

1: pushstack(ARRAY(Code) from VAL2)
2: VAL1 = popstack

end if

end if

end if

end if

end

end

end

end

end

end

end

end
{ let new Dad = value
 { let new Dad = value
 begin
 if value is cons and value is cons then
 and else
 load count
 TRUE else FALSE } push stack (if CODE) if top = EA then
 { begin
 if value is unit and value is unit then
 and else
 load count
 } push stack (if RESULT = true then TRUE else FALSE)
 value = value else
 if value is logic and value is logic then
 load count
 } push stack (if RESULT = true then TRUE else FALSE)
 value = value else
 if value is char and value is char then
 load count
 } push stack (if RESULT = true then TRUE else FALSE)
 value = value else
 load count
 } push stack (if RESULT = true then TRUE else FALSE)
 value = value else

INCYCL

begin

end

<table>
<thead>
<tr>
<th>101</th>
</tr>
</thead>
<tbody>
<tr>
<td>284</td>
</tr>
<tr>
<td>321</td>
</tr>
<tr>
<td>461</td>
</tr>
<tr>
<td>561</td>
</tr>
<tr>
<td>681</td>
</tr>
<tr>
<td>621</td>
</tr>
<tr>
<td>721</td>
</tr>
<tr>
<td>821</td>
</tr>
<tr>
<td>921</td>
</tr>
<tr>
<td>121</td>
</tr>
<tr>
<td>181</td>
</tr>
<tr>
<td>141</td>
</tr>
<tr>
<td>211</td>
</tr>
<tr>
<td>251</td>
</tr>
<tr>
<td>291</td>
</tr>
<tr>
<td>331</td>
</tr>
<tr>
<td>371</td>
</tr>
<tr>
<td>431</td>
</tr>
<tr>
<td>491</td>
</tr>
<tr>
<td>531</td>
</tr>
<tr>
<td>571</td>
</tr>
<tr>
<td>631</td>
</tr>
</tbody>
</table>
case code (prepare op) of
begin
  stack := popstack
end

begin
  if stack.top <= 0 then
    case cont.class (throw, "compare")
    of
      "default": default
      "none": begin
        save cont. once
      end
    end
    begin
      "none": case sub cycle of
        procedure graph block
      end
    end
  end
end

load cont if code(infix op <= 0) then false else true
begin
  pushstack if code(infix op = 0) = true
  the fails must return this
  or error on comparison
  throw := popstack since this is result of false
begin
  stop the recursion (comparing)
if (VTRACK.top) is suspended then
  procedure once block
  end
  cont at CODE at "none"
begin
  coroutine "none"
  end
  procedure none block
  end
  i.e. not prefix

  else (pushstack; Case 1 load cont)
  INCLUDE
  \cont at CODE at "compiler"
  if Trace is suspended then

  Trace := initial CODE prefix opp (Trace)

  if res = arrl LISTER res
  then if res list cont and res = res do

  res = arrl (NEG res)
  else
  res = num (res, the num)
  if res is num then

  res = arrl (NOT, res)
  else
  if res = FALSE then res := TRUE else
  if res = TRUE then res := FALSE else
  res := HEAD res
  end

  defaulj
let rator = popstack

save_cont( "once" ) \ come back here

cont_at( coerce( rator ) , "coarse" )

INCYCL

end else

let rator = popstack

if rator is strict or rator is cons then

begin

pushstack( rator )

end else

if rator is closure then

if rator( fn_def ) is map then

begin : a single def clause

pushstack( rator ) : for error report

pushstack( rator( fn_def )( form ) )

let rand = if CODE( e2 ) isn't suspended

then suspended( CODE( e2 ) , ENV , false )

else CODE( e2 )

pushstack( rand )

ENV := defn( APPLY cons( rator , CODE( e2 ) ) ,

rator( fn_env ) , nil )

save_cont( "binding done" )

cont_at( CODE , "binding" )

INCYCL

end else \ multiple clauses

begin

let arg = if CODE( e2 ) isn't suspended

then suspended( CODE( e2 ) , ENV , false )
let claus = rator( fn def ( clauses )
let this try = claus( hd )
let arglist = link( rator( fn def ( args so far ) ),
               arg )

pushstack( rator )
pushstack( arglist ) ; in case need of another try
pushstack( claus( tl ) ) ; throwing away the hd if no
match or partial found
pushstack( this try )
pushstack( arglist )

ENV = defn( APPLY . cons( rator , CODE( a2 ) ) ,
            rator( fn.env ) , nil )
save cont( "try done" )
cont at( CODE , "try" )
      ! comeback with
INCYCL
end else
    ' is not closure

end else
    if rator is try
begin
    pushstack( closure( rator , ENV ) )
    cont at( CODE , "once" ) ; ' is "once" once more
    INCYCL
end else
    ' not a m/c cycle

end else
    if rator is binding.err
begin
pushstack( rator )
load cont
end else
{ pushstack( err2( APPLY , rator , CODE( a2 ) ) )
load cont
2076 12
2077 11 end
2078 10
2079 9
2080 8
2081 7 procedure got( cntr val ' bool )
2082 6 if STACK( . top ) = val or STACK. rest . top ) =
2083 5 else false
2084 4
2085 3
2086 2 procedure arith. block
2087 1 case SUB.CYCL of
2088 0 "none."
2089 18 begin
2090 17 nodes = nodes + 1
2091 16 pushstack( nil )
2092 15 spawn CODE( a1 ), ENV, STACK. "right"
2093 14 pushstack( suspended( CODE( a2 ), ENV, false ) )
2094 13 cont at( CODE, "data. wait" )
2095 12
2096 11 end
2097 10
2098 9 "data wait"
2099 8 begin
2100 7 let a = STACK( top ) let b = STACK( rest . top )
2101 6 case true of
2102 5 a is er. val b is er. val
2103 4 begin
2104 3 a = popstack b = popstack
2105 2 pushstack( err2( CODE( parop ), b, a ) )
2106 1 load cont PS( lhs. set ) = false
2107 0 end
2108 19 data wait
2109 18
val then true
"data wait"

if enough info do logic

if ( CODE(infix.op) = PAROR and got(TRUE) ) or
( CODE(infix.op) = PARAND and got(FALSE) ) then

let a = popstack
let b = popstack
pushstack( if CODE(infix.op) = PAROR then TRUE
else FALSE
)

load cont

PS(lhs.set) = false / don't expect to be clocked up

else

if isval(STACK(top)) and isval(STACK(rest.top)) then

let a = popstack
let b = popstack
if a is not logic or b is not logic then
pushstack(err2(CODE(infix.op), a, b)) else
pushstack(if CODE(infix.op) = PARAND then TRUE else FALSE)
load cont

PS(lhs.set) = false

else

do monitoring and/or spawning

monitor

default: write "NEVER"

procedure monitor

let a = STACK(top); let b = STACK(rest.top)
if ~(a is nowhere and b is nowhere) do
if a is suspended then
if isval(b) then
<STACK(top) = b spawn(a, nil, STACK(rest), "right")>
else if b is nowhere
if PS(spawnon) and late do
  <spawn(a:nil,STACK("left"),let adj = b(child.size)
  &child.size) = adj+not b(child.spawnon) = true
else a is not suspended
  if b is suspended then spawn(b:nil,STACK(rest),"right")
else
  if b is not nowhere do PS(lhs.set) = true
  te b is val

OUTCYCL :=

procedure act_reg(cpnrtr p)
  let act.ps = 0
  let i = p
  repeat (if i(sub.cycles) = "data wait" do act.ps = act.ps+1
  : = i(next) )
  while i= p
  ps.no(ps.ptr) = act.ps:ps.ptr = ps.ptr+1
  samp1: = samp1+sml
  if psptr > 20 do
    <ps.ptr = i;for i=1 to 20 do
      output d.f: ps.no(i),"","ps.no(i):=0}
  end of my stuff

procedure system( cpnrtr input.exp, input.env )
begin
  procedure processor( cpnrtr ps )
if ps(c) is main or ps(c) is dead or ps(sub_cycle)="dead" then
kill(ps) else
begin
  load context
  PS = ps
  STACK = ps(s)
  ENV = ps(E)
  CODE = ps(c)
  DUMP = ps(d)
  SUB_CYCL = ps(sub_cycle)
  RES_SLOT = ps(res_slot)
  STACKDEPTH = ps(stackdepth)
  CELLS = ps(cells)
  a kick
  only if it is active
  if SUB_CYCL = "data_wait" do
    if CODE isn't overwrite and CODE isn't coarse and
    CODE isn't Print and
    "CODE is infix and CODE(infix.op)=APPLYb) do
    sizeup(PS)
    SIZE = ps(size) ; may have been side-effected
    by sizeup
iterate on number of kicks?

INCYCL
while incycle do
begin
  evalconc
end

I save the context

ps(s) := STACK
ps(E) := ENV
ps(c) := CODE
ps(d) := DUMP
ps(sub.cycle) := SUB CYCL
ps(res.slot) := RES SLOT

* no need to save size, taken care by
ps(stackdepth) := STACKDEPTH
ps(cells) := CELLS

end

output of f, "'nspawn when no of cycles
not = ready
output of f, "'n"
sizeup
let first = process:

stack( nil, nil, 0 ),
input env,
input exp,
dump( Print, "print", nil,
      dump( main, "dummy", nil, nil ),
      "none"
)
stack( nil, nil, 0 ), ; a dummy since the first
      ps does not need one

true
nil, ; sends its result nor left or right
@ i of pntr [ nil, nil ],
nil,
nil,
false, ! lhs not set
C ' size
C ' cells
C ' stack depth
C

first( next ) = first

first( father ) = first

! chain it in
procedure spawn( cpntr code , env , slot , cstring sub.cy )
if isval( code ) and code isn't nothere and
code isn't err .val do
case true of
  code is const :
    slot( top ) = code( the .const )
  code is coarse and code( the susp , its .env ) = EVALUATED ,
  code is suspended and code( its .env ) = EVALUATED :
    slot( top ) = if code is coarse then code( the susp , its .val )
        else code( its .val )
  code is coarse and code( the susp , lock ) ,
  code is suspended and code( lock ) :
    slot( top ) = if code is coarse then code( the susp )
        else code
    sub cy "=" "right" and "late"
    if code isn't coarse and code isn't suspended then
      slot( top ) = suspended( code , env , false ) else
    slot( top ) = if code is suspended then code
        else code( the susp )
default
  begin
    let env = if code is suspended then code( its .env )
        else env
let code = if code is suspended then code else code
else code
let newps = process(
  stack( nil, nil, O ),
  env, code,
  dump( dead, sub. cy, nil, nil ),
  if sub. cy = "right" then "none" else
  if sub. cy = "coarse" then "none" else
  if sub. cy = "left" then "none" else
  sub. cy,
  slot
  if sub. cy = "right" then false else true ,
  if sub. cy="left" then LEFT else RITE ,
  | of ptr( nil, nil ),
  PS,
  nil,
  false
  if sub. cy="right" then PS(size) else O,
  0, ! cells
  if sub. cy="right" then PS(stackdepth) else
  O,
  0, ! lock
)

if sub. cy = "right" then j := j+1 else i := i+1
slot( top ) = nowhere( newps )

! make room in data dep
PS( data dep, 2 ) = PS( data dep, 1 )
PS( data dep, 1 ) := newps
1    chain it in the ring
2    newps( next ) := last in ring( next )
3    last in ring( next ) := newps
4    last in ring := newps

end

procedure find( ps ring -> pntr )

begin
let wanted = ring
repeat wanted := wanted( next )
while wanted( next ) = ring
wanted

end

procedure find father( pntr ps -> pntr )

if ps( next ) = ps then ps else

begin
let ptr = ps( next )

let stop := false
repeat<
for i = 1 to 2 do
if ptr( data dep , i ) = ps do stop := true

while stop = false do ptr = ptr( next )

ptr

end

end
procedure have_child( cptr ps ) -> int

let child = ps( data dep )
if child[1] = nil then 0 else
  if child[2] = nil then 1 else
    2

procedure unlock( cptr ps )

if ps( c ) is overwrite do
  ps( c, susp, lock ) = false
let dmp = ps( d )
while dmp ~= nil do
  if dmp( Code ) is overwrite do
    dmp( Code, susp, lock ) = false
  dmp = dmp( Dump )

procedure kill( cptr ps )

if ps ~= nil do
  if have_child( ps ) = 0 then
    begin
      if ps( c ) is main do ring = nil
      if ring ~= nil do
        (
          ! unchain it
        let previous = find ps( ps )
}
previous | next | = ps( next )
if last in ring = ps do last in ring = previous
let p = find_father( ps )
let i = 1
while p: data dep , i ) = ps do i := i+1
if i = 1 do p( data dep , 1 ) = p( data dep , 2 )
p( data dep , 2 ) = nil
if ps(line) = RITE do
{ p(cells) = p(cells) + ps(cells)
p(stack depth) = ps(stack depth) }
}
if messages and ps(c) is main do
begin
output of
"ncomputation completes"n",
"nits size in m/c steps",
"size = " , ps( size ),
"nmaximum stack depth = " , ps( stack depth ),
"ncells used = " , ps( cells ),
"nlock cycles = " , ps( lock.cycles ),

"n"
"ntotal locks = " , GLOCK
"n ************************************"n",
"n"n"
end
unlock ps
end else
for i = 1 to have. child ps do kill ps( data. dep , i )

procedure lock wait
procedure declar (pntr ds, env -> pntr)
begin

procedure declar (pntr form, expr, guess, oldenv -> pntr)
if form is id then
  if expr is suspended then defn (form, expr, oldenv)
else
  defn (form, suspended (expr, guess, false), oldenv)
else
  if form is const or form is repetition then oldenv
else
begin if form is cons (a list)
  let com. expr = if expr is suspended then expr else
    suspended (expr, guess, false)
  let hdcode = suspended (prefix (HD, coerce (com. expr)), nil, false)
  let tlcode = suspended (prefix (TL, coerce (com. expr)), nil, false)
  env = declar (form (hd), hdcode, guess, oldenv)
  env = declar (form (tl), tlcode, guess, env)
end
procedure show(string message)
end

else any( LEFT.DEFIN )
endif

when any = null and any (DEFN name) = null do

while any = null and any (DEFN name) = null do

begin

begin

begin

begin

begin

begin

begin

begin

begin

begin

begin

begin

begin

begin

begin

begin

begin

begin

begin

begin

begin

begin

begin

begin
begin
  errorflag := true
  message.given := true
  write "new " , message . " in or near 'n"
end

procedure err1 (pntr op , arg -> pntr )
  error( prefix( op , arg ) )

procedure err2 (pntr op , arg1 , arg2 -> pntr )
  error( infix( op , arg1 , arg2 ) )

procedure print(pntr v)
  ' part of the evaluator now. ie a m/c instruction
begin
  case true of
  v is suspended
    begin
      save cont( "print" )
      cont.at( coerce( v ) , "coerce"
    end

  v is char
    output o.f : v( the . char )

  v is num
    output o.f : v( the . num )

  v is logic
    output o.f : v( the . bool )
pushs tacx(T1)

save_cont("print"); the T1
pushstack(Hd)

end

v is closure function id; v = true

default
begin
output o.f., "Illegal Expression."
show(v)
activity = notgoing; a flag to print
incycle = false; exit the main process
end

if have output an object

if v is logic or v is num or v is char or v = nil or v is closure do

begin
flush(o.f.)
load_cont
end
procedure show(pntr v)
if v is const then show(v( the const )) else
if v is id then output of v( the id ) else
if v is num then output of v( the num ) else
if v is char then
  output of case v of
    NL : "nl"
    SP : "sp"
    NP : "np"
    TAB : "tab"
  default : "%" ++ v( the char ) else
if v is logic then output of v( the bool ) else
if v = nil then output of "()" else
if v is cons then
  begin
    output of "(" repeat
    { show(v( hd ))
      output of " "
      v = v( tl )
    } while v is cons
    show(v)
    output of ")"
  end else
if v is cond then
  begin
    show(v(test))
    output of " -> " show(v(left.fork))
begin
output x.f, "a"
end

if v is closure then function id(v, true) else

if v is suspended then show(v, its val?) else

if v is strict then output o.f, v( strict.op )

if v is repetition then show(v( the rpt )) else

if v is map then

begin
output o.f, "a"
let v = v
while # is map do

begin

show( f( form ) )
output o.f, "=)"

f := f( body )

end

show( f )

output o.f, "a"

end else

if v is true then

begin

let t := v( clauses )
while t = nil do

begin

show( t( hd ) )
output o.f, "\n"

end

end else

if v is er val then show( v( arg ) ) else

if v is prefix then show prefix(v) else

if v is in-fix do

case v( infix.op ) of
begin
output o f , "(" 
show( v( e1 ) )
output o f , ""
show( v( e2 ) )
output o f , "")

end

begin
show( v( e2 ) ) \"the expression\"
output o f , " where \"\nshowenv( v( e1 ) , 4 )
output o f , "1"
end

begin
showenv( v( e1 ) , 4 )
output o f , "1"
end

begin
output o f , "(" 
show( v( e1 ) )
output o f , ""
show( v( e2 ) )
output o f , "")
end

procedure show prefix( zptr v )
case v( prefix op ) of

begin
output o f , "List Wanted: <"
show( v( e ) )
output o f , ">"

end
CDND
begin
  output o.f., "?C
  show(v(e))
  output o.f., "? -> ?"
end

UNDEF
begin
  output o.f., "Undefined:";
  show(v(e))
  output o.f., "?"
end

default
begin
  output o.f., v(prefv op show(v(e))
end

procedure showenv(pntr env; int count)
begin
  let i = count
  let ds := env
  while i > 0 and ds /= nil do
    begin
      if ds(defn.name) /= APPLY do
        begin
          output o.f., "\n"
          show(ds(defn.name))
          output o.f., " = "
          show(ds(its defn))
          i := i - 1
        end
( mnemonic ), " "
procedure function id( pntr f ; bool brackets )

begin
  procedure equiv( pntr object , definition -> bool )
  ! object is a closure which we are comparing with definition in env
  if definition is suspended and definition( its val ) is closure then
    object = definition( its val ) or ! i.e. same closure
  begin
    let obj = object( fn def ) ! map or trys
    let def = definition( its val , fn . def ) ! map or trys
    obj = def or
    obj is trys and def is trys and
    begin
      ! is first clause for obj a clause of def ?
      let obj . first = obj . clauses , hd
      let defs = def . clauses
      while defs != nil and obj . first != defs . hd do
        defs = defs . 1
      end
      obj . first = nil
    end
  end
  else false ! definition cannot be a function "looked up" to produce object

procedure find( pntr obj , env -> pntr ) ! opposite of lookup
begin
  let env = env
while env = nil and 'equiv obj', env( its. defn ) do
  env = env( next. defn )
  if env = nil then nil
  else env( defn. name )
end

! function id

let name = find( f, the. env )
if brackets do output of f, "C" ! open top level only
if name = nil then show( name )
else begin
  let f. env = f( fn. env )
  let name = find( f, f. env )
  if name = nil then show( name )
  else begin
    ' get partial mapping
    let ap = lookup( APPLY, f. env )
    if ap = nil then output of f, "Function id error"
    else begin
      function. id( ap( hd ), false )
      output of f, ""
      show( ap( tl ) )
    end
  end
end

if brackets do output of f, "D" ! close top level only

end

' main program