ANOTHER IMPLEMENTATION TECHNIQUE FOR APPLICATIVE LANGUAGES

Hugh Glaser and Sean Hayes

1 ABSTRACT

Data flow has sometimes been proposed as an evaluation mechanism for applicative languages, its focus on data dependency and the pure functional nature of certain models can make it an attractive choice. Few machines have been built however, and those that have fall short of being ideal general-purpose applicative language computers. This paper will present a particularly simple data flow model which is similar to supercombinator reduction, supporting higher order functions, garbage collection and a form of lazy evaluation in a clear and natural manner.

The paper will also show how the model can be made to execute on a conventional processor. Such a method has shown significant speed increases over other available methods of evaluating functional programs, and the hardware implementation holds the promise of a machine that executes applicative languages at a comparable speed to conventional hardware executing control flow programs. In addition, unlike other proposed models, the extensions to a multi-processor machine are natural and well defined, with the potential of even greater speedups.

The work reported in this paper was carried out at:
Department of Computing, Westfield College London and then King's College London
Both authors are now at:
Department of Computing, Imperial College, 180 Queen's Gate, London, UK

ANOTHER IMPLEMENTATION TECHNIQUE FOR APPLICATIVE LANGUAGES

2 INTRODUCTION

In the search for efficient implementations of functional languages, researchers have looked at a number of alternatives to the traditional von Neumann execution mechanism [Vegdahl]. One of the aims of using a different approach is to take advantage of the lack of side effects in functional languages. Reflection of this characteristic in the underlying architecture should allow a good method of distribution of computation amongst the processors of a multi-processor machine. Data flow models have been proposed as suitable evaluation mechanisms to exploit parallelism, and much work has been done on both models and associated architectures [Kahn] [Treleaven et al.].

The work in the Department of Computing at King's College London (then at Westfield College) started with the design of a theoretical data flow model and proceeded with the specification of both the textual functional language and the graphical programming system which is naturally associated with it [Hankin] [Hankin & Glaser]. It was expected that any architecture design that resulted from this work should be well orientated towards functional languages and furthermore have a sound theoretical basis.

One such architecture has been developed [French & Glaser] and a first prototype machine, known as the TUKI, built. The work reported here is a method of implementing a refinement of the architecture by software emulation, which we call the C-TUKI.

The paper begins with a discussion of the underlying model of acyclic, once-used data flow and describes how instructions derived from this notation can be implemented on a serial machine. A software emulation of the machine is described which makes it possible to efficiently compile applicative languages for conventional architectures. The final section of the paper reports on some of the results obtained, and directions for future research.

3 THE ABSTRACT MACHINE

In this section we will describe the data flow model and show how it is being used to implement supercombinator reduction. The primitive operations of the model are described, and we look at how a partially ordered graph may be converted into a linear stream of instructions. These instructions are then shown to be a basis for stack based execution of applicative programs. Finally we give a complete description of the abstract machine in the form of pattern matching rewrite rules.

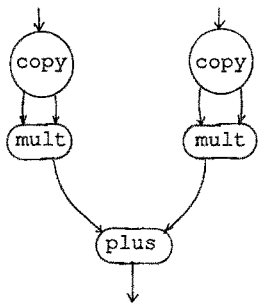
3.1 Data-flow Models and Supercombinators

The notion of data flow is based on the concept of a network of processes connected by data paths. In a purely functional data flow model the processes act solely on the data arriving on their input data paths, and the data that is sent on the output data paths is no more than a function of the input data. Data flow programs are usually represented as two dimensional pictures, where boxes or circles represent processes or functions and arrows represent data dependencies.

For example, the lambda expression:

```
λa.b. +(* a a)(* b b)
```

is represented as a data flow graph by:

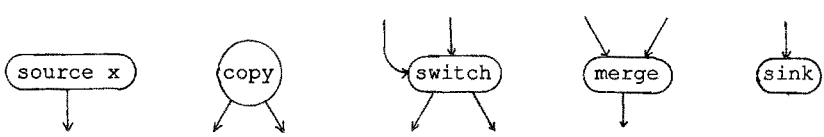


In the model used here, which is based on [Hankin], the graphs are restricted to be acyclic, that is the input of a process can never be affected by its output. As a result of this restriction the graphs are once used, in that only one datum traverses any particular data path, and when a process has sent data on its outputs it ceases to exist. The graphs are dynamic, which means that they can be extended as functions are called, by inserting a copy of the graph of the called function into the graph being executed. Such data flow graphs are just another way of representing lambda expressions, and in effect form a machine code for arbitrary combinators.

Combinators are simply lambda expressions with no free variables [Curry & Feys], the prefix "super" coined by Hughes, is used to distinguish the general class from the S, K, I family due to Curry and described by Turner [Turner 1979]. Although a form of the lambda calculus, restricted to combinator expressions, could be used directly as a programming language, it is more convenient to use a richer programming language, fo example Hope [Burstall et al.] or Miranda [Turner 1984], and convert programs to the simpler form by removing free variables and nested lambda expressions [Hughes]. The simpler form can then be converted to the data flow machine code.

3.2 The Primitive Operations

The primitive operations of the data flow model fall into three groups. The first of these are the pure data routing primitives.



The simplest is the source node, with no inputs and only one output, which is used to introduce constants into the graphs. These are drawn from the integers, booleans, and other primitive types. It can also produce function identifiers which are the names of data flow graphs. Identifiers may be bound to some parameters and later expanded into their defining data flow graphs, with the parameters placed on the input arcs.

The copy node has one input and reproduces the value that appears on the input on both output arcs.

The switch is used to control the flow of data, particularly in conditional expressions where it acts as a guard preventing the passage of parameters into the unselected

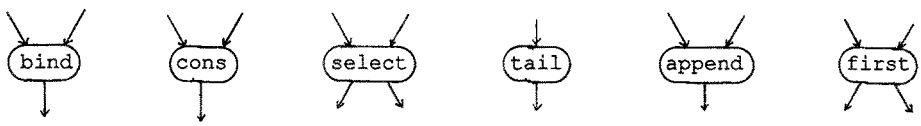
branch of the conditional. The left input of a switch is evaluated to obtain a boolean, if the value is true the left output arc is selected to receive the value from the other input, if the value is false the right output is selected. The unselected arc is left undefined.

The merge, unlike that in the formal model, is uncontrolled and selects one of its inputs to be sent onto the output arc. The node is deterministic however, selecting the right input only if the left input has no input value to use. This is the only node that can produce output if one of its inputs is undefined, all others, including the bind operation described below, have all outputs undefined whenever any input is undefined.

In the physical realisation of the model, arcs that are to be left undefined actually carry the data value KILL, which allows the merge to act deterministically.

The sink is a notional operator that takes one or two inputs and has no outputs, this node is sometimes used as the destination for switch nodes in graph diagrams to clarify which output of the switch is being used when the other is not required.

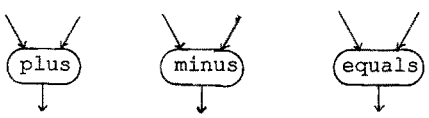
The second group are the structure forming operations.



The most important is the bind operation which takes two inputs: the left one is a functional, either a function identifier or the result of a previous bind, the right is a parameter to be bound to the functional. The result is another functional which may later be expanded into its defining graph. In the model all functions are treated in a curried fashion and are bound to one argument at a time. Higher order functions are a natural result, as the mechanism for collecting parameters can be used to store excess arguments. The excess are used when the first function returns a function which requires them.

The other operations in this class are the list and stream (including i/o) operations and are not specified further.

The third group of operators are the usual primitive functions like plus, minus, equals, etc..



These operators are strict in both their operands. This means that when presented with function bindings, they cause them to be converted to data flow graphs and evaluated to basic values.

Execution of graphs in this model proceeds in a data-driven, rather than demand-driven, fashion but this presents no obstacle to supporting lazy semantics: since the values flowing along arcs may be unevaluated function bindings or recipes for building structures. These are evaluated as far as necessary only when they enter a strict operator (from the third group, or the left input of a switch node). This allows us to support lazy lists and streams, and gives a fine control over the evaluation time of parameters.

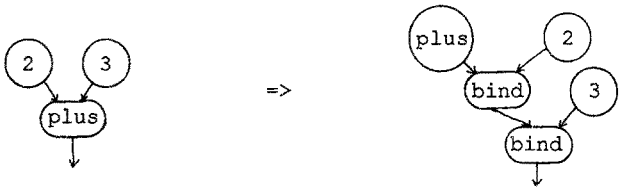
For example a safe eager evaluation mechanism can be achieved by inserting a strict identity node just after switches controlling parameters. In the expression:

```
f(1,bigfunction(a,b,c))
where
        f is [x,y]
        if condition-involving-x
                then ...y...y...y
                else 0
        endf
wend
```

Strictness analysis [Mycroft] cannot show at compile time that y is definitely required. But by putting a strict identity after the switch that controls y it will be evaluated as soon as the condition is found to be true, so that only copies of the value of y, rather than pointers to a suspension, are propagated.

A completely lazy system is achieved by reducing the set of strict operators to a single need operator, treating the rest as two place functions.

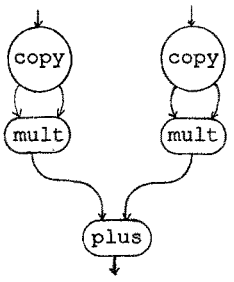
For example:



3.3 Partial to Total Ordering

In the graphs seen so far the data dependencies define a partial ordering and not the total ordering of instructions required by serial machines. In a multi-processor machine this property is advantageous in exploiting potential parallelism, as two operations not dependent on each other can be executed concurrently. For the single processor machine we are considering however, the graphs must be converted into a totally ordered form.

For example:



sumsq

```
is converted  to:

copy  2.left 2.right
copy  2.left 2.right
mult  2.left
mult  1.right
plus
```

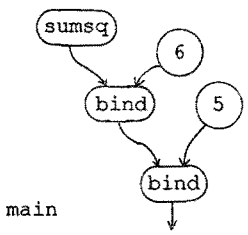
The numbers represent offsets from the current instruction, the modifier `.left` or `.right` determines whether they are the left or right input values of the target instruction.

There are two possible methods of conversion using tree searching algorithms (ignoring duplicate visits); the first is a depth first pre-order (or post-order) tree walk, the second is a reversed, breadth-first, analysis. Both maintain the property that data flows in one direction only, allowing sequential execution, but the pre-order walk produces the best code for serial machines and is the method used here, with the extra condition that nodes dependent on the parameters appear before any others in the program.

In the example above a true pre-order walk would put the second `copy` instruction after the `mult` instruction, but the machine requires that nodes dependent on the parameters must appear first.

3.4 Stack Based Execution

This section illustrates the steps in the execution of a function `main` which calls the `sumsq` function above after binding parameters 6 and 5 to it.



The program is put onto the program stack; an instruction on this stack consists of an opcode and two relative offsets from the instruction into the data stack, these offsets are the slots into which the result of the instruction will be placed. The data stack contains two fields which contain the inputs for their corresponding instruction, the program stack and data stack run in lock step so that the top entry of each can be paired up to execute.

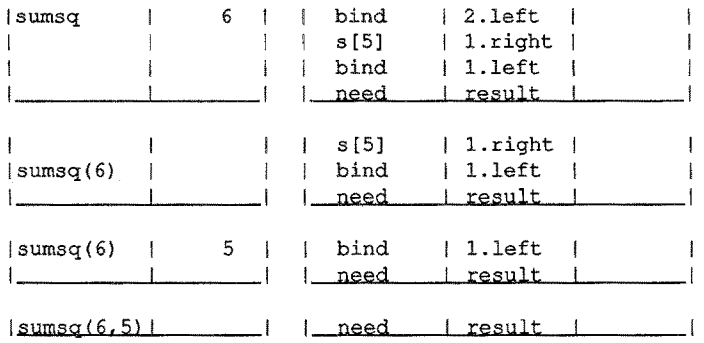
Initially the data stack consists only of "holes" waiting for data from the preceding instructions, and the program stack contains only the instructions for `main`, and the instruction which caused the `main` function to be started (represented here by `need`).

`s[x]` represents a source instruction for `x`.

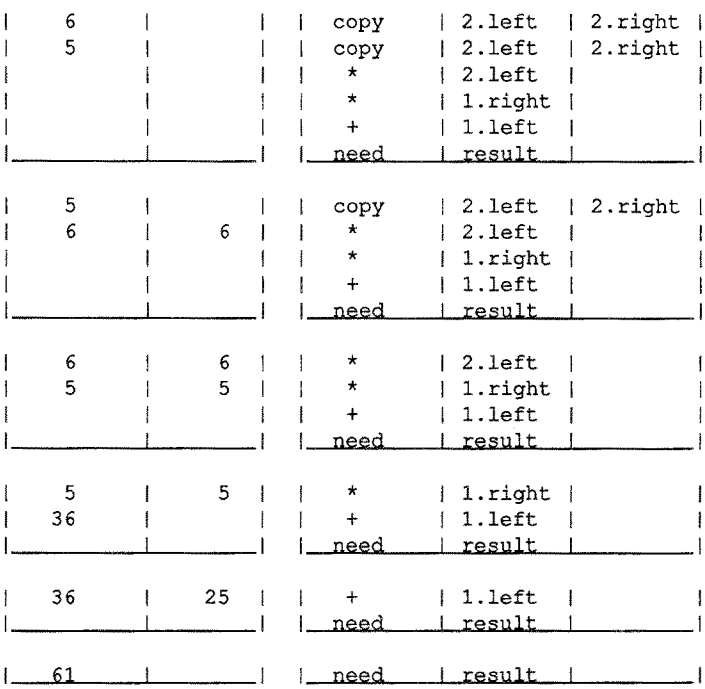
```
          data stack                    program stack

|           |         |   | s[sumsq] | 2.left  |         |
|           |         |   |  s[6]    | 1.right |         |
|           |         |   |  bind    | 2.left  |         |
|           |         |   |  s[5]    | 1.right |         |
|           |         |   |  bind    | 1.left  |         |
|___________|_________|   |__need____| result  |_________|

|           |         |   |  s[6]    | 1.right |         |
|sumsq      |         |   |  bind    | 2.left  |         |
|           |         |   |  s[5]    | 1.right |         |
|           |         |   |  bind    | 1.left  |         |
|___________|_________|   |__need____| result  |_________|
```



The function `main` returns the fully bound functional `sumsq(6,5)`, which is then the input for the `need` instruction. This instruction is assumed strict and therefore causes the function binding to be evaluated. The code for `sumsq` is stacked onto the program stack, the bound parameters are put into the left inputs of the first two data stack entries and execution begins again.



The `need` instruction finally has the value which it can use.

In the example above we have shown function bindings being entered into the data stack, in an abstract description this presents no problem, but there are two good reasons why in a real machine bindings should be represented as references to another, heap based, memory organ called the function or bind store. The benefits are that only fixed sized objects are stored in the data stack, and more importantly, should a binding ever be copied it need only be evaluated once, as the property of referential transparency allows the binding to be overwritten by its value, allowing other references to the same binding to simply pick up the computed value.

Since it is impossible to create cycles in the bind store we can conveniently use reference count garbage collection. Garbage collection issues are ignored in the abstract

presentation that follows, where we assume an infinite number of possible function store entries.

3.5 The Abstract Machine Description

The abstract machine is best described as a triple (P, D, F), where P represents the instruction stack, D represents the data stack and F represents the function or bind store.

In the description that follows:

d_i represent places in the data stack.
a, b ... represent arbitrary data values.
v_i represent basic data values.
o represents a "hole".
T_i are the names of places in the function store.
I_i are arbitrary instructions.
$D[d_i \backslash v]$ represents the stack D with the value v at place d_i.
$F+\{T:a\}$ represents the value a stored at place T in the function store.

The instructions are presented in the form of pattern matching rewrite rules, no rules are given for the list and stream operations as they are generally understood and would not add to the understanding of the basic machine.

The primitive operations are represented by the general operator op. The entries in the function store are tagged with bind, func or id. Where the tag is func, the value is the representation of a data flow graph. The numbers represented by m and n are the number of instructions and number of parameters expected respectively.

Abstract Machine Description

((source a d_1).P, (o, o).D, F) => (P, D[d_1\a], F)

((sink).P, (a,b).D, F) => (P, D, F)

((copy d_1 d_2).P, (a,o).D, F) => (P, D[d_1\a][d_2\a], F)

((switch d_1 d_2).P, (T_1,a).D, F) => ((call T_1).(switch d_1 d_2).P, (T_1,a).D, F)
((switch d_1 d_2).P, (true,a).D, F) => (P, D[d_1\a], F)
((switch d_1 d_2).P, (false,a).D, F) => (P, D[d_2\a], F)

((merge d_1).P, (o,b).D, F) => (P, D[d_1\b], F)
((merge d_1).P, (a,o).D, F) => (P, D[d_1\a], F)
((merge d_1).P, (a,b).D, F) => (P, D[d_1\a], F)

((bind d_1).P, (T,b).D, F) => (P, D[d_1\T_2], F+{T_2: bind T b})

((op d_1 d_2).P (T,b).D, F) => ((call T).(op d_1 d_2).P, (T,b).D, F)
((op d_1 d_2).P (a,T).D, F) => ((call T).(op d_1 d_2).P, (a,T).D, F)
((op d_1 d_2).P (a,b).D, F) => (P, D[d_1\(op a b)][d_2\(op a b)], F)

(call T_1).P, D, F+{T_1:(bind T_2 a)}) => ((call T_2).P, (a,T_1).D, F+{T_1:bind T_2 a})

((call T).P, (a_1,T_1)...(a_n,T_n).D, F+{T:(func m n I_1...I_m)) => (I_1...I_m.(return).P,
(a_1,o)...(a_n,o).$(o,o)_{n+1}$...$(o,o)_m$.(o,T_n).D, F+{T:(func m n I_1...I_m)})

(call T).P, (T,b).D, F+{T:(id v)}) => (P, (v,b).D, F+{T:(id v)})
(call T).P, (a,T).D, F+{T:(id v)}) => (P, (a,v).D, F+{T:(id v)})
(call T_1).P, (T_1,b).D, F+{T_1:(id T_2)}) => ((call T_2).P, (T_2,b).D, F+{T_1:(id T_2)})
(call T_1).P, (a,T_1).D, F+{T_1:(id T_2)}) => ((call T_2).P, (a,T_1).D, F+{T_1:(id T_2)})

((return).P, (a,T_1).D, F) => (P, D, F+{T_1:(id a)})

4 IMPLEMENTATION

A library package of routines has been written in the 'C' programming language and works with the UNIX operating system. This package enables suitably formed TUKI assembler programs to be compiled and run on conventional hardware.

This is a portable and fairly efficient compromise between writing an interpreter, such as the Miranda system, and compiling to native assembler code like the G-machine project.

The G-machine [Johnsson] is another architecture that uses a method similar to the one described here, but is based on a graph reduction machine code instead. Both the architecture presented here and the G-machine use directly programmed sequences of reduction, and so are closer to traditional computer models than more radical designs, that modify the code and compute the next reduction at every step. This makes it easier to use existing technology to build the machines.

The C-TUKI package consists of a number of short routines, each of which implements a single TUKI instruction. These in turn call procedures to implement the data-stack and bindstore.

4.1 Of The Program

A typical assembler function (`sumsq` again) looks like:

```
sumsq()
{       _copy(dsp-4,dsp-5);
        _copy(dsp-2,dsp-3);
        _mult(dsp-4,X);
        _mult(dsp-3,X);
        _plus(dsp-2,X);
        return;
}
```

Each line of the body of the function is a call to one of the library procedures. Looking in more detail, the code for the `plus` instruction is:

```
#include <stdio.h>
#include "all.h"

_plus( OUTL, OUTR)

register DATASTACK *OUTL, *OUTR;
{   IF killtest()
          THEN
                OUTL->type=KILL;
                OUTR->type=KILL;
          ELSE
                force(INL);
                force(INR);
                (OUTL->value).intval =
                (OUTR->value).intval =
                      (INL->value).intval + (INR->value).intval
    ENDF;
    pop();
}
```

The `killtest` predicate at the start of the first instruction tests to see if either input to the instruction is the special `KILL` value. Every instruction (except `merge`) does this; if either input is `KILL` the entire instruction is ignored and `KILL` is sent to both outputs. If the non-kill input is a reference to the bindstore then the object pointed to is de-referenced. It is this reclamation of references to the bindstore from the datastack, which forces the implementation to explicitly kill instructions rather than use jumps to go around them. It is possible to avoid this inefficiency, but we explain here the method of implementing `KILL` that bears the closest correspondence to the data flow model.

If the predicate allows the instruction to continue, the next test is to see if either input is a bind (this happens in strict inputs only). If it is then, as in the abstract model, it must be forced to a simple value. The procedure `force` performs all of the work in increasing the size of the datastack, fetching the parameters to the data-stack and calling the function. Nesting of function calls and returns are managed by the standard 'C' stack and are not explicitly programmed.

4.2 Of The Datastack

The information required by each instruction is always at the top of the datastack. the variables `INL` and `INR` are pointers to the left and right input values respectively. The `pop` procedure is called by every instruction, which removes the top pair from the data-stack. The parameters `OUTL` and `OUTR` are passed to the instruction and are computed as offsets from `dsp` the current top of the data-stack.

4.3 Of The Bindstore

As we have said, functions are applied to parameters one at a time by the bind instruction, which in the C-TUKI looks like:

```
#include <stdio.h>
#include "all.h"

_bind( OUTL, OUTR)

register DATASTACK *OUTL, *OUTR;
{
    BINDCELL * newcell();
    register BINDCELL * hold;

    IF killtest()
          THEN
                OUTL-> type= KILL;
                OUTR-> type= KILL;
                hold = newcell(); /* reference count at 1 */
                hold->ltag = INL->type;
                hold->rtag = INR->type;
                hold->head = INL->value;
                hold->tail = INR->value;
                OUTL->type = BIND;
                (OUTL->value).ptr = hold;
```

```
                IF OUTR != X
                      THEN /* Add the extra data-entry & references */
                            OUTR->type = BIND;
                            (OUTR->value).ptr = hold;
                            reference(hold);
                            if( INL->type == BIND)
                                  reference((INL->value).ptr);
                            if( INR->type == BIND)
                                  reference((INR->value).ptr);
                ENDF;
    ENDF;
    pop();
}
```

This uses the function `newcell`, which accesses a free slot in the bindstore. This is eithe a cell freed when its reference count becomes zero, or if there are none with zero count, the next top of heap. The bindstore is allocated in a series of linked pages by the standard UNIX memory allocator. Calls to the allocator can be kept to a minimum because the bindstore can be reference count garbage collected, this is one of the majoi advantages over graph reduction schemes.

The method actually used is the "lazy garbage collection" method of [Glaser & Thompson] where de-references are not acted upon immediately, but placed on an intermediate stack. Future claims for a free cell take the top entry off this stack perform the de-reference. If the count is left at zero then the cell is immediately re-used, if not the action is repeated with the new top of stack. This method is memory efficient in machines that store the heap on slow remote memory and have a fast cache-type memory available to store the de-reference stack. This is because it cuts down one read-modify-write cycle per cell reclaimed. In our case however its major advantage is that there are no long waits as memory is reclaimed: even if large structures are being reclaimed, only the head is claimed and the two sub-pointers are pushed onto the stack. Another potential advantage is to have a second processor, or idle times of one, to reorder the stack to bring several de-references to the same cell together, and minimise page faults.

5 CONCLUSIONS

We have timed this implementation using the "nfib" benchmark, beloved of functional programmers, that gives a figure for the number of function calls per second. In comparison with combinator based implementations on the same hardware the C-TUKI ran between two and three times faster. This would not compare favourably with the reported figures for the G-machine, however we have only implemented a naive version of the C-TUKI. It is expected that a fully optimised version, for example using jumps and also macros to replace function calls, would give a significant increase in performance. In addition, if native machine code was generated directly, as in the G-machine, we would hope to achieve similar timings.

It can be seen that the method descibed here is a viable tool for the implementation of applicative languages. It is expected that hardware development using these principles will produce a powerful computing engine. We are currently implementing a second prototype and simulation of the multi-processor version (the m'TUKI) has given encouraging results.

6 REFERENCES

Burstall, R.M., MacQueen, D.B. and Sanella, D.T., HOPE: An Experimental Applicative Language, Proc. 1980 Lisp Conference, Stanford 1980.

Curry, H.B. and Feys, R., Combinatory Logic, Volume 1, North-Holland 1968.

French, E.F. and Glaser, H.W., TUKI, A Data Flow Processor, Computer Architecture News 11, 1, 12-18 (March 1983).

Glaser, H.W. and Thompson, P., Lazy Garbage Collection, Westfield College, Internal Report.

Hankin, C.L., A Data Flow Model of Parallel Processing, Department of Computer Science, Westfield College, University of London, Aug. 1979, Ph.D. Thesis.

Hankin, C.L. and Glaser H.W., The Data Flow Programming Language CAJOLE - An Informal Introduction, ACM SIGPLAN Notices 16, 7, 35-44 (July 1981).

Hughes, J., Graph Reduction with Supercombinators, PRG-28, Programming Research Group, Oxford University, 1982.

Johnsson, T. The G-Machine: An Abstract Machine for Graph Reduction, Proceedings of the Declarative Programming Workshop, University College London, April 1983.

Kahn, G., The Semantics of a Simple Language for Parallel Programming, Information Processing 1974 (IFIP 74), North-Holland, 1974.

Mycroft, A., Polymorphic Type Schemes and Recursive Definitions, Proc. 6th. International Conference on Programming, Toulouse (Springer-Verlag LNCS 167) April 1984.

Treleaven, P.C., Brownbridge, D.R. and Hopkins, R.P. Data -Driven and Demand-Driven Computer Architecture, Computing Surveys 14, 1, (March 1982).

Turner, D.A., A New Implementation Technique for Applicative Languages, Software - Practice and Experience 9, 1, 31-49 (Jan. 1979).

Turner, D.A., Miranda: A Non-Strict Functional Language with Polymorphic Types, Proc. Conference on Functional Programming and Computer Architectures (Springer-Verlag LNCS 201) 1984.

Vegdahl, S.R., A Survey of Proposed Architectures for the Execution of Functional Languages, IEEE Transactions on Computers, C-33, 12 (Dec. 1984).