Evaluating Functional Programs on the FLAGSHIP Machine

Paul Watson Ian Watson*

Abstract

The Flagship project has the aim of designing a parallel computer system for the evaluation of declarative languages. The physical architecture of the machine consists of a set of closely coupled processor/store pairs communicating over a high performance communications network. A functional program to be evaluated is compiled into a Super-Combinator expression graph which is then simplified by graph reduction. The paper discusses the Flagship machine architecture and describes in detail the computational model which defines how a functional program is represented and reduced. The issues underlying the design of the main features of the computational model are examined.

1 The Flagship Project

The Flagship Project is a collaboration between teams working at the University of Manchester, Imperial College London and International Computers Ltd. Its aim is the production of a parallel computer system for the evaluation of declarative languages. Work is being carried out in areas which include: parallel computer architectures, declarative languages, persistent storage environments, systems software and, computational models for the evaluation of declarative languages. This paper discusses only part of the project: the architecture of the parallel machine and the computational model for the evaluation of Functional Programs.

2 The Parallel Machine Architecture

The Flagship machine architecture has been designed to perform Graph Reduction [PEYT87]. Programs are represented as graphs, which are simplified to give the result of the program. By simplifying different sub-graphs simultaneously, parallel graph reduction machines offer the possibility of reaching the result of the program more quickly than is possible with a serial machine. Whether or not this can be achieved in practise depends on the machine architecture, the computational model which governs how the graph is simplified, and the program itself (a serial program cannot be executed more quickly on a parallel machine

*Authors' Address: Department of Computer Science, The University, Oxford Road, Manchester, M13 9PL, U.K.

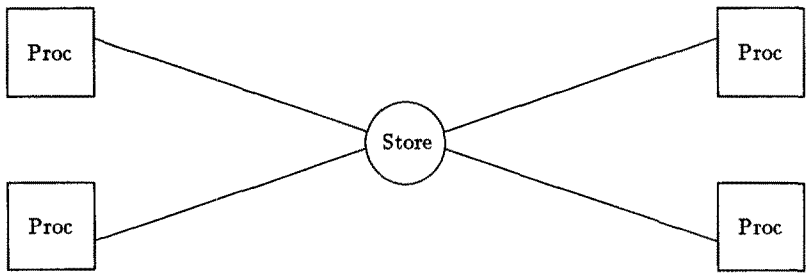


Figure 1: The Abstract Architecture

than on a serial machine). The abstract machine architecture is shown in Fig. 1. It consists of a single store which can be accessed by a set of processors. The program to be evaluated is compiled into a graph (the Computational Graph) in the store. This is then simplified by graph reduction, with all the processors potentially able to simplify different sub-graphs in parallel.

The physical architecture consists of a set of closely coupled processor/store pairs connected together by a high performance communication network (Fig. 2) [WATS87a]. The single store of the abstract architecture has been divided among the processors, each of which can directly access only that part of the store to which it is closely coupled. All accesses of a non-local store must be achieved by sending a request message to the processor coupled to the store. Each processor simplifies those reducible sub-graphs contained in its local store. The architecture is designed to be extensible: the power of the machine can be increased by adding more processor/store pairs.

The main reason for employing this form of physical architecture is so that there is no shared memory to become a bottleneck in the system. If all processors had to access the computational graph in a shared store (as in the abstract architecture) contention for the store would seriously degrade the performance of the machine. The physical architecture also allows the vast majority of store processor interactions to take place between a processor and its local store, over a very high bandwidth, low latency link, rather than through the necessarily slower communication network.

The physical architecture raises the issue of how the computational graph is spread among the stores of the processors. In order for all the processors to work in parallel, each must have at least one reducible sub-graph in its local store. How can this be achieved ?

One possibility is to carefully partition the computational graph over the processors before evaluation begins. This is impractical, as the number and distribution of the reducible sub-graphs in the computational graph varies dynamically as an evaluation proceeds. For most programs, soon after the evaluation began, the distribution of reducible sub-graphs would bear little relation to the initial distribution.

We believe that a mechanism is needed to dynamically move reducible sub-graphs between the processors' stores as an evaluation proceeds. Such a 'load-balancing' system has been devised by Sargeant

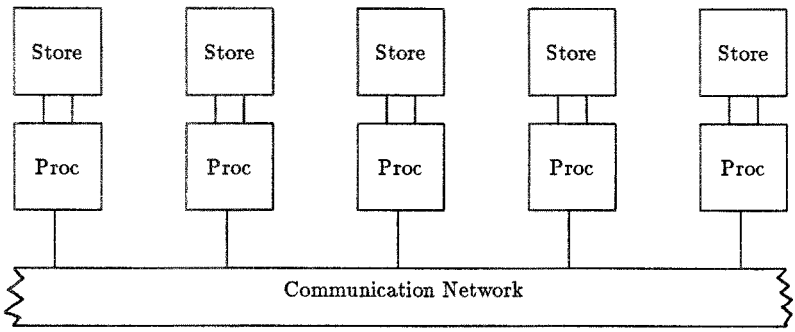


Figure 2: The Physical Architecture

[SARG86]. It attempts to keep all of the processors in the machine equally active by moving reducible sub-graphs from busy to idle processors during the evaluation of a program. A full description of the system is outside the scope of this paper, but simulation results suggest that it is very effective, and so it will be used in the Flagship machine.

Another problem created by this form of architecture is how to deal with a reducible sub-graph which is not fully contained within the store of a single processor [WATS86a]. One solution would be to allow the processor reducing the graph to access non-local stores by sending messages through the communication network during reduction. The latency of these messages would be very high compared to the time it takes to access the local store, and so would seriously degrade the speed of reduction. The scheme used in the Flagship machine is to store in the root of each reducible sub-graph information which allows the machine to bring together all of the sub-graph into a single processor before rewriting is attempted. This ensures that during a rewrite, all store accesses are to the processor's local store. As autonomous hardware connected to each processor can organise the gathering together of reducible sub-graphs into a single processor's store while it is reducing another sub-graph, the performance of the machine is to a large extent independent of the latency of the network, so long as there are enough reducible sub-graphs available to keep all the processors in the machine busy.

Serial portions of computation do occur even in programs which may have a high average parallelism, and it is for this reason that the network is designed to have a high performance. We do not want the serial parts of a computation to become a serious bottleneck in program execution.

3 Representing the Computational Graph

Before it can be evaluated, a functional program must be compiled into a graph. Every storage location in the machine can hold a single node (called a packet), each of which is globally addressed. A packet consists of a Header field followed by a number of Item fields (packets are provided in the store with a variety of numbers of Item fields). The Header holds information about the whole packet, such as how many Item fields it has, and what state of reduction it is in. One of its sub-fields, the Packet Type, denotes what the packet represents; for example a function application or a constructor.

An Item field can hold an atomic constant, such as an Integer, or a packet address, allowing graphs of packets to be represented.

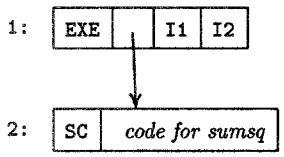
The computational model takes a program represented as a Super-Combinator [HUGH82] expression graph and reduces it to Weak Head Normal Form (i.e. until there is no top level redex) [PEYT87]. Earlier in the Flagship project, experiments were performed simulating computational models based on fixed (Turner) combinators [CHEE85][TURN79] and the Lambda Calculus [WATS86b]. The results of these investigations suggested that the Super-Combinator model is between ten to a hundred times more efficient because it increases the granularity of the reductions, and reaches the result in fewer reductions. Although the amount of computation required in each of the reductions may be larger, there is an overhead in accessing the graph to be reduced and writing back the result of the reduction, which makes it more economic to perform fewer, larger grained reductions than more small grained ones. It is of course important that the granularity is not increased so much that the amount of exploitable parallelism is decreased [GOLD85].

The Super-Combinators themselves are compiled into a conventional sequential imperative machine code which, when executed, takes the arguments to which the Super-Combinator is applied and constructs the graph which represents the result of reducing the application. The imperative code compiled from a Super-Combinator is held in a packet in the expression graph. Unlike all the other packets, these Super-Combinator packets have no Item fields; instead the Super-Combinator code is stored where the Item Fields would normally be. An application of a Super-Combinator to its arguments is represented by a packet, whose first Item field holds the address of a Super-Combinator packet, while the other Item fields hold the actual parameters (or their addresses if they are not atomic constants).

For example, the functional program (in HOPE [BAIL85]):

```
dec sumsq: num X num -> num;
--- sumsq(a,b) <= a*a+b*b;
sumsq(1,2);
```

would be compiled into the graph:



Each packet is shown preceded by its address in the packet store (arbitrary addresses have been used). The only Header field which has been shown is Packet Type (EXEcutable application and Super-Combinator), and this is then followed by either the Item fields (I1 denotes the integer 1), or the imperative code in the case of the Super-Combinator packet. The **Executable** packet type denotes the reducible application of a Super-Combinator to arguments. To reduce the graph, the processor executes the imperative code defining **sumsq**, which performs the arithmetic and writes the following graph back into store:

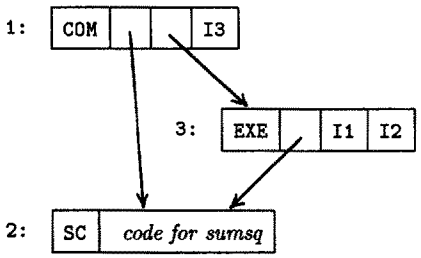
The packet type **CONstructor** is used to denote that the packet can be used to hold atomic values (as in this case) or to form data structures (this is discussed in section 7).

One feature of the computational model is the storing of the Super-Combinator code in packets in the expression graph. Most other designs for machines to evaluate declarative languages, for example the G-machine [KIEB85], have a separate Super-Combinator code store. This is loaded up with the code before the program is evaluated, and in the program graph a Super-Combinator is referred to by its address in that special store. We are however interested in the design of a general purpose machine which will allow multiprocess computing. We do not want to have to devise mechanisms to allow new Super-Combinator code to be loaded (and removed) from a separate store while other programs are evaluating. The synchronisation problems in doing this would be too great. Also on a more philosophical note, in functional programs, functions are treated as first class citizens. We feel that this is achieved if they are represented in the program graph with all other types of expressions. It enables functions which generate code (compilers) to be written in a natural way. In fact, the Super-Combinator code packets can even be lazily generated as they are required !

Another advantage of this approach is in the support it gives for those programs (usually for A.I. applications) in which functions are created and modified dynamically during the run time of a program. In a model, such as that presented here, in which the function definitions exist in the program graph, modifying a function can be achieved by overwritting the Super-Combinator packet compiled from it, with its new definition.

The Super-Combinator code is written to assume that its strict arguments are already evaluated. In the case of sumsq(1,2) where both arguments are strict, this is true, but if the expression sumsq(sumsq(1,2),3)

was encountered during the course of a program evaluation, how would it be reduced? The outer **sumsq** application cannot be reduced until sumsq(1,2) has been reduced to an integer. In this case, the expression would be represented by the graph (again the packet addresses are arbitrary):



When a function application is compiled, if the strict arguments of the function will not definitely have been evaluated when the application is reduced then a packet of type **COMplete application** is used for the application, rather than one of type Executable. The Header of a Complete Application packet contains a field which holds information about the strictness of the function being applied. When the packet is to be reduced, this information is used to check whether or not all the strict arguments of the Super-Combinator have been evaluated. If they have, then the code in the Super-Combinator packet can be executed as in the previous example. If this is not the case, then the strict arguments are evaluated before the Super-Combinator code is executed.

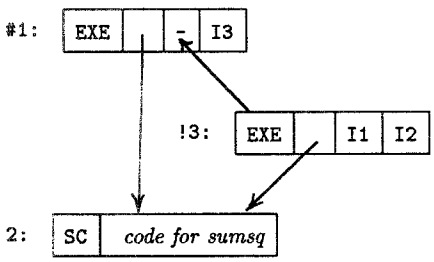
4 The Execution Mechanism

The mechanism by which graphs of packets are reduced is designed to be data-driven [WATS85]. Graphs of packets only become considered for reduction when some progress can be made. When a graph is reduced, the result of the reduction is sent to all those other graphs waiting for the result. Its arrival makes them candidates for reduction. This section describes how this is achieved.

The Header of every packet contains a Packet State field. This takes one of three values: Dormant, Active or Suspended. Initially, all packets are in the Dormant State. When an expression is to be reduced, the Packet State field of the root packet of the expression graph is set to Active. Each processor keeps a list of the addresses of all the packets in its local store whose state is Active (the Active Packet Queue). The processor's basic action is to pick an address from that queue and reduce the graph rooted at that address. First it reads, from a field in the root packets header, the information as to the sub-graph which must be in the local store before reduction can begin. By restricting the size of the maximum sub-graph which can be reduced in a single step to the root packet and its direct descendants, this information can be encoded into a small field in the header. If some of the packets required are not in the local store then requests are

sent through the network for copies of those packets, while the root packet is suspended until all the copies have returned. A cache in each processor is used to reduce the amount of re-copying of non-local packets. If however all the sub-graph is in the local store, then the processor can proceed to reduce it. To do this it examines the Packet Type of the packet at the root of the graph, and then acts accordingly. In the case of an Executable packet, as has been seen, it accesses the Super-Combinator packet addressed by its first Item field, and executes the code held in it, constructing the graph which is the result of the application. The Executable packet is then overwritten with the result of the application so that it can be shared.

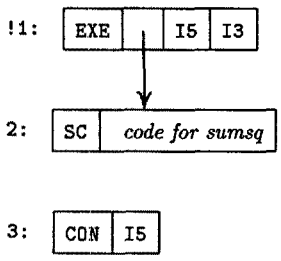
For a Complete Application packet, the processor's action is different because it must check that the strict arguments have been evaluated before the Super-Combinator code can be executed. First of all, the strictness information in the Header of the packet is examined, and a check is made as to whether or not the strict arguments of the application are evaluated. If they all are, then rewriting can proceed immediately as for an Executable packet. If however any of them are not, the execution of the Super-Combinator code must be delayed until they have been evaluated. In the previous example, this is the case, and so the following graph is written back to the store:



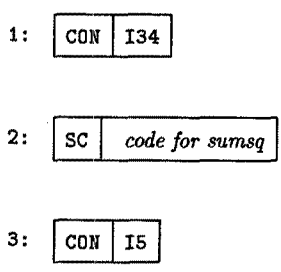
In the above, the packets address is prefixed by a representation of the value of its Packet State field: (# : Suspended, ! : Active, no annotation : Dormant). The Complete Application packet at address 1 has now had its Packet Type changed to Executable, and its state to Suspended. This denotes that the packet cannot be evaluated until the evaluation of some of its Item fields has been completed (in this case the second). The packet at address 3 has had its state changed to Active because it is a strict argument which must be evaluated. When a packet has been made Active, another Item field is used to hold the address to which the result of its evaluation should be sent. This is denoted in the above diagram by the arrow from the packet at address 3 to the Item field to which the result of evaluation should be returned. This is called the Return Address. It is possible that another part of the computation will already have made the graph active. In this case, a list of Return Addresses is built from the Return Address Item field.

When the packet at address 3 was made Active, its address would have been added to its local processor's Active Packet Queue, and sometime later this address would have been picked from the queue by the processor, which would then have rewritten the sub-graph rooted at address 3. In this case, because the

root packets type is Executable, the Super-Combinator packet addressed by its first Item field would be accessed, and the code executed. The result of this would be the graph:



The Executable packet at address 3 has been overwritten with a Constructor packet holding the result of its evaluation so that it can be shared. The result has also been returned to the Item field specified by its return address (the second Item field in the packet at address 1). Because of this, the packet at address 1 is not now waiting for any of its arguments to be evaluated, and so its Packet State has been changed to Active. The processor can now rewrite the graph at address 1 to give:

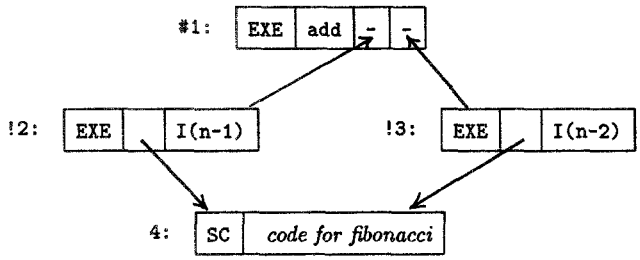


This shows why rewriting a Complete Application packet produces a suspended Executable packet. When the Executable packet becomes Active, its strict arguments will have been evaluated, and so the Super-Combinator application can be performed.

Strictness information can often be used to allow the compiler to generate more efficient code by allowing Executable packets to be planted, instead of Complete Application packets. For example, when the following function is compiled:

```
dec fibonacci : num -> num;
--- fibonacci(n) <= if n < 2
                    then 1
                    else fibonacci(n-1) + fibonacci(n-2);
```

the Super-Combinator code generated would first test if n was less than 2. If it was not, it would overwrite the application fibonacci(n) with the graph:



This graph assumes the application packet was stored at address 1.

The argument I(n-1) denotes that the imperative code performs the subtraction of 1 from the argument n, and places the integer result in the Item field. The packet at address 1 is suspended, and cannot be reduced until two arguments (the packets at addresses 2 and 3) have been evaluated. So that the return of one of these arguments does not cause the packet to become active again, the Header of every packet contains a Suspended Count Field. When a packet becomes suspended, this field is set to the number of Item fields being evaluated. When an evaluation is completed and the result returned to an Item field in the suspended packet, the Suspended Count Field is decremented, and only when it becomes equal to zero is the packet's state changed to Active.

The packet at address 1 contains a built in operation in its first Item field: **add**. There are a small set of built in operations provided to perform arithmetic, logical and selection operations. The application of a built in operation to arguments is performed exactly as for a Super-Combinator application, except that the code defining the application is known to the computational model, rather than being held in a Super-Combinator packet.

We have taken the decision that the code compiled for Super-Combinators will assume that all strict arguments have already been evaluated. The Complete Application packet is used to ensure that this is the case. It would however have been possible to compile Super-Combinator code which checked whether the arguments on which it was strict were evaluated, and forced their evaluation and suspended the application if they weren't. This is the approach taken in the ALICE machine [DARL81]. It removes the need for Complete Application packets: only Executable application packets would be needed, however one advantage of using the Complete Application packet to force the evaluation of strict arguments is that it reduces the number of store accesses. The Super-Combinator packet is not accessed, and no code within it is executed, until the application can be reduced because the strictness information is held in the Header of the Complete Application packet. Another advantage is that the code compiled for a Super-Combinator is shorter. It does not have to test if the strict arguments are evaluated and create a suspended application packet if some of them aren't.

One question arising from the example evaluation of the fibonacci function is why the result of the reduction, including the two active packets, is written back into the graph store. Why not continue reducing the graph, perhaps on a stack as in serial evaluators [KIEB85] except for sending to other processors enough active packets to keep the whole machine busy. There are several reasons for writing the graph back into store after each reduction step.

Firstly, it allows the load-balancing system to take full control of the parallel tasks in the system. It can decide to move the active packets created by a rewrite from busy to idle processors. It can also control the amount of parallelism in the machine. Managing the Active Packet Queue in a First In First Out regime tends to create parallelism by evaluating the computational graph in a breadth first manner, while managing it as a Last In First Out stack tends to restrict the parallelism but utilises less graph storage. The load balancing system alternates between the two regimes depending on the total number of parallel tasks in the machine: FIFO is used to create tasks to keep all the processors busy, LIFO to restrict the store utilisation when all the processors are busy. Stack based evaluation corresponds to a FIFO Active Packet Queue, which is undesirable when parallelism needs to be created.

If the computation is allowed to take place on a stack with occasional tasks being created and exported to other processors as required to keep the machine busy, then sometimes the stack will have to be suspended because its execution cannot proceed until a result has been received back from another processor. Saving an arbitrarily large stack may present problems of storage management, but a more serious problem is that after the de-suspension of a stack, only a small amount of evaluation may be possible because a result being evaluated by another processor is again required. This will require the stack to be suspended again, and if it is large the relatively high overhead of de-suspension and suspension compared to the actual amount of evaluation will become significant.

Another problem in allowing large tasks to build up on a stack is that they cannot be decomposed into parallel tasks. For example, the following scenario might occur in a stack based evaluator. Because there are enough parallel tasks in the system to keep all the processors busy, a processor performs a large evaluation on its stack without writing anything back into the graph store, or exporting any tasks. After some time, the stack begins to collapse to the result of the evaluation, but also many of the other processors begin to run out of work to do because they have completed evaluating their stacks and now require new tasks to occupy them. While the processor with the large collapsing stack has a lot of work to do, it cannot decompose the stack back into parallel tasks, and so the machine becomes unbalanced, with some processors idle while one has a large amount of work to do. This scenario cannot occur in the packet based model presented in this paper as the graph structure is always maintained, and so it is possible at any time to export tasks, whether the graph is expanding or contracting.

We believe that packets are the correct logical unit of the expression graph to suspend and de-suspend. Stacks are of arbitrary size when they are suspended: their size depends on the state of the computation

which has been reached when an event causes the suspension. In effect, we are using packets to hold small stacks whose size is carefully chosen, so that only closely related parts of the expression graph are suspended and de-suspended together. When a graph is evaluated, the execution mechanism transforms it into what can be viewed as a two dimensional stack. Simulation results suggest that this method of graph reduction is comparable in efficiency with the fastest serial stack based evaluators and also has the advantage of naturally supporting parallel evaluation [WATS87c].

Another reason for not using a stack is that the Flagship machine must support multiple processes with multiple priorities. This is done by assigning a priority to each active packet. At each reduction step, the processor selects the highest priority active packet whose address is in the Active Packet Queue. This means that reductions from different programs and processes will be freely intermixed, and so with a stack based scheme, suspension would happen frequently, considerably reducing its effectiveness.

Another advantage in keeping all rewrites distinct is that we want to use the Flagship machine to support several different types of declarative languages. We are not convinced that all of the languages that we want to support will fit into a stack based reduction strategy, and so we believe that to design a machine around a stack may reduce its generality.

5 General Function Application

The use of Executable and Complete Application packets to evaluate the application of a Super-Combinator to arguments is limited to a special case of application. That is when the compiler plants the code for the application of a known Super-Combinator to a number of arguments equal to its arity. In general Super-Combinator expression reduction, there are cases where the Super-Combinator being applied is unknown at compile time. For example, the function **map** which takes a function and applies it to each element of a list can be written in HOPE as:

```
typevar alpha,beta;
dec map: (alpha-> beta) X list(alpha) -> list(beta);
--- map(f, []) <= [];
--- map(f, h::t) <= f(h)::map(f,t);
```

The Super-Combinator code implementing the second equation must generate a packet to apply f (the function being mapped) to an element of the list. However, it is not known at compile time whether or not the function will be strict on its argument, and so the strictness field in the Header of a Complete Application packet could not be set correctly.

Another problem concerns the arity of a Super-Combinator. For example, if **g** is a Super-Combinator of arity 2, then the application of **g** to a single argument must not cause the Super-Combinator code to be executed to perform the application, because **g** has not been applied to a number of arguments equal to

its arity. This is one of the basic rules of Super-Combinator reduction.

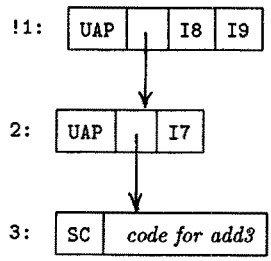
A new set of packet types is provided to support these more general applications. The Executable and Complete Application packet types are actually unnecessary - all applications could be compiled to graphs of the more general packet types. However, Complete Application and Executable are important optimisations:- for first order function applications they allow compile time information to reduce run-time computation.

6 The General Model

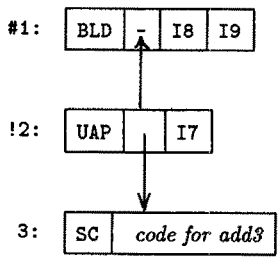
In the general model, applications are compiled using **Unevaluated Application** Packets. The rules used to reduce these packets will be described with reference to the example Super-Combinator application:

```
((add3  7)8 9)
```

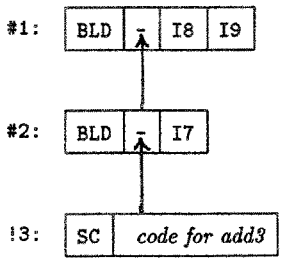
where **add3** is a Super-Combinator which adds together three integers. This expression would be compiled into the following packets:



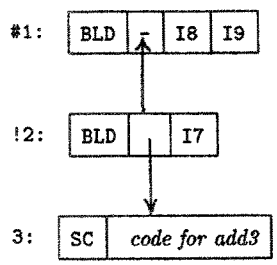
When an Unevaluated APplication packet is reduced, it forces the evaluation of its first Item field, and is rewritten as a suspended **Build** packet. This forces the function to be evaluated before it is applied to arguments. Thus the graph would now become:



The same reduction rule applies when the Unevaluated Application packet at address 2 is reduced, giving:



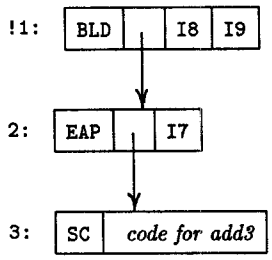
The Headers of Super-Combinator packets contain fields holding their arity and information as to on which arguments they are strict. In this example, the arity of **add3** is 3, and it is strict on all three of its arguments. When an active Super-Combinator packet is reduced, there are 2 possibilities. If its arity is 0 then the Super-Combinator code is executed. If however it is greater than zero, then the packet cannot be reduced, and so its address is returned to the Item field specified by the return address. This is what happens in this case, giving:



When a Build packet is reduced, the packet addressed by its first Item field is accessed. If it is a Super-Combinator packet then its arity field is examined. If the arity is equal to the number of arguments in the Build packet, then the application can be reduced. The packet type is changed to Executable, and any unevaluated strict argument packets are made active. The required strictness information is obtained from the Header of the Super-Combinator packet.

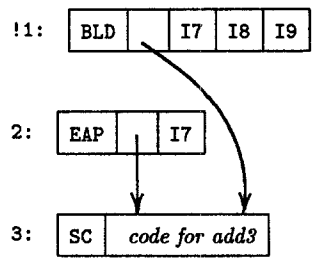
Alternatively, if the arity of the Super-Combinator is greater than the number of arguments in the Build packet (as in the example), then the application cannot be reduced. The Build packet becomes a

dormant **Evaluated Application** packet and the address of the packet is sent to its return address:



The Header of the Evaluated Application packet has an arity field which is filled in with the arity of the result of the application: in this case 2 as another 2 arguments are needed before the application can be reduced.

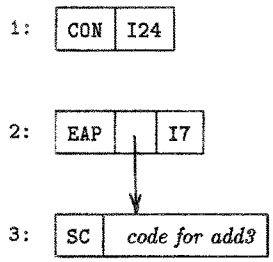
When a Build packet is reduced, if its first Item field addresses an Evaluated Application packet then the two application packets are combined into a single application packet. This happens in the example evaluation:



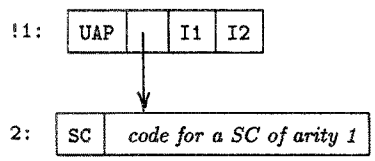
The arity of this new application packet can be determined by subtracting the number of new arguments applied (i.e. those in the Build packet originally at address 1) from the arity of the Evaluated Application packet (at address 2). If the arity of the combined application is greater than zero, then the combined application has its type set to Evaluated Application as it cannot be reduced.

If however the arity of the combined application is zero, then it is a reducible application, and its first Item field will hold the address of a Super-Combinator packet. The strictness of the Super-Combinator is accessed from the Header of the packet, so that any strict arguments can be evaluated. If this is necessary, then the application is stored as a suspended Executable packet. When it becomes active again the application will be reduced. In the example, all the strict arguments are already evaluated, and so the

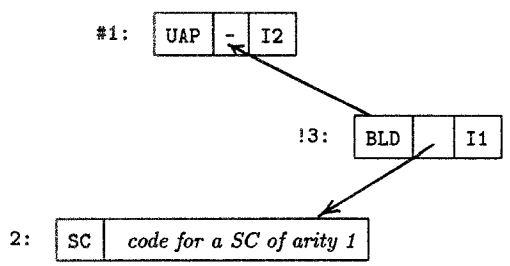
reduction can be performed immediately, producing the graph:



There is one other possibility for Unevaluated Application packets. Their first Item field may hold the address of an Evaluated Application, or Super-Combinator packet whose arity is less than the number of arguments in the Build packet. For example:



In this case the graph is rewritten as an application of the number of arguments equal to the arity of the Super-Combinator, returning to the first Item field of an application packet holding the remaining arguments:



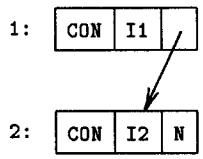
7 Constructors

The examples have shown how Constructor packets are used to hold the results of evaluations, by overwriting an application with a Constructor packet holding the result of its evaluation. The Flagship Machine uses Weighted Reference Count Garbage Collection [WATS87b], which gives enough information to determine, at run time, whether or not the result of the application is shared. If it is not, then the planting of

the Constructor packet is unnecessary and so is not performed.

This is one advantage of using a form of Reference Count garbage collection, rather than a Mark-Scan scheme, as in the latter it would be impossible to tell if the result of the application was shared, and so a Constructor packet would have to be planted each time.

Constructor packets are also used to represent data-structures. For example, the list [1,2] would be represented as:



N is a type of Item field representing the empty list. When a Constructor packet becomes active, it rewrites in one of two ways. If it contains a single Item field, then just that field is sent to the return address. If it has more than one Item field, then it must send its own address to the return address. Lazy data-structure evaluation is achieved by not evaluating the arguments of the Constructor packet when it becomes active.

Sometimes it is desirable to allow data structures to be evaluated eagerly in order to create parallelism. To support this, another packet type **Unevaluated Constructor** is provided. Its Header contains strictness information which is used to force the evaluation of some or all of its arguments, when it becomes active. Once they have been evaluated it becomes a dormant Constructor packet.

8 Other Computational Models

The previous sections have shown how the functional programming computational model is implemented by compiling programs to graphs of packets. The packet types then determine how the computational graph is rewritten to produce the result of the program.

The Flagship machine was designed to be a general purpose machine, not just one for evaluating functional programs, and so it is important to consider how other computational models may be implemented. There are two methods by which this can be achieved. The first is by defining Super-Combinators to reduce graphs in a way appropriate to the model. For example, a logic programming model may use a Super-Combinator which unifies together two terms. The second method is to define new packet types which perform the reductions appropriate to the model. This is potentially far more efficient as complex reduction schemes can be represented by a packet type, rather than being held in Super-Combinator packets which are then executed.

We would like people to be able to design models of computation (including new packet types and their

associated rules) without requiring them to have a detailed knowledge of the hardware and firmware within the Flagship machine. In order to allow this, computational models can be specified in the high level graph reduction language DACTL [GLAU87]. This can then be used by an engineer familiar with the Flagship machine to write the firmware implementing the reductions corresponding to the packet types used in the model. A considerable proportion of the time spent in designing the Flagship reduction mechanisms has been expended in ensuring that they are both powerful and flexible enough to implement these DACTL specifications.

By this method, a Lambda Calculus expression reducer has been designed, while work is proceeding on parallel LISP and parallel PROLOG computational models.

9 The Present State of the Project

The functional programming model of computation is currently being evaluated on a software simulator. A parallel processing facility is being built, and will be completed by early 1988. It will consist of about ten 68020 processor/store pairs communicating initially over a VME bus. In time, a delta network will replace the VME bus, considerably increasing the communication bandwidth.

The parallel processing facility will allow far larger functional programs to be run than is possible on a software simulator. We hope to extensively evaluate the proposed machine architecture and model of computation on this facility, and so collect the detailed information needed to design the custom parallel processing machine which is one of the major goals of the Flagship project. It is expected that the final machine will utilise custom hardware for the processor, and will also provide hardware support for common operations such as storage management and garbage collection.

10 Acknowledgements

The authors would like to thank their colleagues in the Flagship project for many useful ideas and interesting discussions. We would also like to thank Ursula Hayes for preparing the diagrams.

11 References

BAIL85 A HOPE Tutorial, R. Bailey, Internal Document, Dept. of Computing Science, Imperial College London, 1985.

CHEE85 The Applicability of SKI(BC) Combinators in a Parallel Rewrite Rule Environment, A.B. Cheese, MSc Thesis, Dept. of Computer Science, University of Manchester, 1985.

DARL81 ALICE - A Multi-Processor Reduction Machine for the Parallel Evaluation of Applicative Languages, J. Darlington and M. Reeve, Proceedings of 1981 ACM Conf on Functional Programming Languages and Computer Architecture, 1981.

GLAU87 DACTL, J.R.W. Glauert J.R. Kennaway and M.R. Sleep, in International Computers Ltd. Technical Journal, Summer 1987.

GOLD85 Serial Combinators, B. Goldberg and P. Hudak, in Functional Programming Languages and Computer Architecture, Lecture Notes in Computer Science, No. 201, 1985.

HUGH82 Graph Reduction with Super-Combinators, R.J.M. Hughes, Oxford University PRG Technical Monograph PRG-28,1982.

KIEB85 The G Machine, R.B. Kieburtz, in Functional Programming Languages and Computer Architecture, Lecture Notes in Computer Science, No. 201, 1985.

PEYT87 The Implementation of Functional Programming Languages, S.L. Peyton Jones, Prentice Hall, 1987.

SARG86 Load Balancing, Locality and Parallelism Control in Fine-Grain Parallel Machines, J. Sargeant, University of Manchester Dept. of Computer Science Report UMCS-86-11-5, 1985

TURN79 A New Implementation Technique for Applicative Languages, D.A. Turner, Software Practice and Experience, Volume 9, 1979.

WATS85 Parallel Data Driven Graph Reduction, I. Watson, P.Watson and J.V. Woods, in Fifth Generation Computer Architectures, ed. J.V. Woods, North-Holland, 1986.

WATS86a Graph Reduction in a Parallel Virtual Memory Environment, I. Watson and P. Watson, in Proceedings of Santa Fe Graph Reduction Workshop, Sept. 1986.

WATS86b The Parallel Reduction of Lambda Calculus Expressions, P. Watson. PhD Thesis, University of Manchester, 1986.

WATS87a Flagship Computational Models and Machine Architecture, I. Watson, J. Sargeant, P. Watson and J.V. Woods, in International Computers Ltd. Technical Journal, Summer 1987.

WATS87b An Efficient Garbage Collection Scheme for Parallel Computer Architectures, P. Watson and I. Watson, in Proceedings of the European Conference on Parallel Architectures and Languages, Eindhoven, The Netherlands, June 1987.

WATS87c The Cost of Parallel Graph Reduction, I. Watson, J. Sargeant, P.Watson, J.V. Woods, FLAGSHIP Project Internal Report, 1987.