AN ARCHITECTURE FOR A LOOSELY-COUPLED PARALLEL PROCESSOR

Robert M. Keller
Gary Lindstrom
Suhas Patil

UUCS - 78 - 105

October 1978

Department of Computer Science
University of Utah
Salt Lake City, Utah 84112

This work was supported in part by grants DCR-74-21822, MCS-77-09269 and MCS-78-03832 from the National Science Foundation.

Abstract: An architecture for a large (e.g. 1000 processor) parallel computer is presented. The processors are loosely-coupled, in the sense that communication among them is fully asynchronous, and each processor is generally not unduly delayed by any immediate need for specific data values. The network supporting this communication is tree shaped, with the individual processors connected at leaf nodes. The machine executes a graphical version of applicative Lisp. The program execution model is demand-driven, with a special deferred interpretation for dotted pair evaluation, termed "lenient cons". Opportunities for concurrency arise in the parallel evaluation of arguments to strict operators, i.e. those known to require evaluation of their full set of arguments. Such opportunities are exploited by exporting function application tasks to neighboring processor nodes in the tree, subject to a hierarchical notion of load balancing. Locality of task allocation and communication is a key objective of the machine. An integrated design toward that end is presented, combining language issues, firm semantic foundations, and anticipated hardware technologies.

keywords and phrases: applicative programming, architecture, concurrency, data flow, demand-driven, lenient cons, Lisp, locality, loosely-coupled, packet switching, parallelism, reduction machine, tagged architecture.

CR categories: 6.21, 4.22, 4.12, 4.32

CONTENTS

1. INTRODUCTION

The architecture of highly-parallel machines has received increased attention from researchers over the past decade. At first, because of their novelty, workers were content with proposing elaborate machine architectures without giving great consideration to how such machines would ultimately be programmed to exploit their available computational power. Experience with Illiac IV, Star-100, etc. has shown this to be a mistake. Indications are that programming languages deserve consideration at the earliest stages of architectural conception. Included in such considerations are issues such as storage management and task management.

This paper describes considerations for what might be called a *loosely-coupled architecture*. This term was used in [Arden and Berenbaum 75] in discussing memory management trade-offs in multi-processor systems. We use it to denote a machine which potentially incorporates a large number (say 1000) of processors which can function independently to a large extent, but which can effectively communicate with one another when necessary. Furthermore, we require that the computations being supported are not tied to the structure of the machine at the program level. A corollary of this architectural concept is that the system is easily *expandable*, there being no logical dependence on the number of processors. Such expandability is further enhanced by the particular physical organization to be described. Additionally, through the use of a packet switching intercommunication network, the system can by seen to have many of the features attending *reconfigurable systems*, (*cf.* [Reddi and Feustel 78]).

The architecture presented here was influenced by work reported in [Dennis and Misunas 74] and [Arvind and Gostelow 77] on *data flow machines*. Our machine architecture attempts to bring internal communication costs within the machine to a more manageable level by taking advantage

of *locality* of reference. The communication network in our machine plays the role of the arbitration and distribution network of the Dennis data-flow machine. However, the processing units which assemble instructions and initiate information flow are more like the processors of Arvind and Gostelow. Even though the architecture of our machine has a tree-like structure, it is not a "recursive architecture" in the sense of [Davis 78]. Our system has in common with those cited in this paragraph the desire to integrate architectural and language considerations. This is one of the ways it differs from superficially similar systems, such as Cm* [Swan, *et al.* 77]. These similarities and differences will be further reviewed in Section 16.

Our architecture is currently in the development stage. We present in this paper some of the major philosophical decisions which are influencing us, along with an execution model for a subset of the ultimate machine language.

2. LANGUAGE ISSUES

Heretofore, research on highly-parallel machines seems to have predominately emphasized numerical, rather than symbolic, computations. We feel that further investigation of the latter is merited. The possibility of such applications has been alluded to before, e.g. [Hearn 76]. Presently we are choosing Lisp as a target language for our architecture. We would like to present arguments in further defense of this choice. The first is that there is a substantial community of Lisp users who are seeking the higher computing speeds which a parallel processing computer can give. We believe that the problem of the acceptance of a new architecture will be substantially solved if Lisp can be supported on the computer, since that choice would not involve acceptance of a new language.

Secondly, we feel that Lisp, possibly with some advice on programming style, can be much better matched to the power of a loosely-coupled system than other languages. For example, extensive transformation of Fortran programs is done to make effective use of the Illiac IV, e.g. [Lamport 74]. Consequently, the connection between object and source programs is obscured, and debugging is affected adversely. We feel that the object language of our machine can be made reasonably close to a usable subset of Lisp.

Furthermore, Lisp, with some minor modifications, such as *lenient cons* discussed later (*cf.* [Friedman and Wise 76], [Henderson and Morris 76]) seems to include all opportunities for exploitation of concurrency that proposed data flow languages do. It also seems to provide more, e.g. concurrent operations on tree or graph data structures during the latter's creation, and natural ways for dealing with conceptually infinite structures.

Finally, even if full Lisp proves to be too difficult to support efficiently, in our attempt to design a machine for it, we will gain valuable experience about the inherent difficulties in supporting such languages on a loosely-coupled computer.

It may seem that catering to Lisp would have the effect of excluding most of the potential users of other data-flow machines, e.g. those interested in large *numerical* computations, as users of our machine. It is our hope that such users will approach our design with an open mind. We believe, for several reasons, that our machine can compete with others in the numerical computation domain. First, although our evaluator is different, other machines are likely to incur very similar mechanization problems, making the execution speeds similar for the same underlying computation, independent of source language used. Secondly, numerical computations, e.g. large Fortran programs, can be mechanically translated into Lisp. There are known case studies, e.g. [Fateman 73], where the Lisp version actually runs faster, even when iteration is replaced with recursion.

3. BASIC ARCHITECTURE

Figure 1 shows the physical arrangement of components in our machine. The internal nodes of the tree structure are bi-directional communication units, thus combining the attributes of the arbiter and distribution units of the Dennis machine along with additional *balancing* functions. Processing units are attached to the machine as leaf nodes. The leaf nodes are not necessarily equidistant from the root node of the tree. One might expect, for example, special-purpose units, of which there are relatively few, to be closer to the root node, for enhanced accessibility and utilization. Although the figure shows a *binary* tree, and the discussion in this paper makes that assumption for simplicity, technology considerations suggest that a 4-ary or 8-ary tree might be more appropriate.

A general processing unit is roughly the size of a conventional micro-computer, but its architecture is substantially different. It is able to carry out local computation, particularly with respect to assembly and dissemination of information, and to initiate actions for fetching information from other nodes of the tree. It will be able to execute single program tasks, and create tasks in response to the execution of *invoke* (procedure application) operations, which may then be executed either in the local processing unit or in another processing unit.

The primary memory of the system is distributed among the processing units. Each processing unit has immediate access to that segment of memory located within it. It also has access, through the *communication network*, to the segments of memory located at other processing units.

Even though the memory is distributed among the processing units, there is only one *unified logical address space*. Given the address of a datum, any node in the machine is able to logically access it directly. The internal nodes of the communication network are responsible for any required physical routing of addresses and data. Access to auxiliary memory and other forms of external communication take place through special-purpose leaf processors.

4. COMMUNICATION NETWORK

The communication network is designed to help the machine to take advantage of locality of information flow, thereby reducing communication costs which often tend to be high in data-flow oriented machines. It is also responsible for distributing the computing load among available processing units.

In the data-flow machine of Dennis, the arbitration and distribution networks are disjoint, and any piece of information which needs to be sent from one instruction cell to another needs to traverse the entire depth of these networks, even if the cells are physically close neighbors. By combining the arbitration and distribution functions, we can cut down the distance information needs to travel in such cases.

In our machine, information first travels up the tree towards the root node until it comes to a node from which the destination cell is reachable by going down the tree, then it proceeds down the tree until it finally reaches the desired destination cell. Thus, for sending or receiving information from neighboring cells, it is not necessary for the information to travel the entire depth of the tree. Relatively local data-flow therefore takes less time and improves the overall communication cost of the computation. Furthermore, another important consequence of combining the arbitration and distribution networks is that the traffic congestion at the narrow ends of these networks is reduced, enabling the communication network to handle a higher volume of data.

A second function of the communication network is to provide a reasonably balanced distribution of the computing load. Such a function is not required in the Dennis machine, as the latter does not attempt to allocate tasks dynamically (i.e. cell addresses are fixed at compile time). Each node of

our communication network periodically obtains *monitoring signals* from its subordinates, which indicate their current utilizations. When such signals indicate a sufficiently unbalanced state, the node can cause the transfer of uninitiated tasks from one subtree to the other (see Section 15).

5. LOCALITY

One of the most important concepts of our architecture is to improve performance by exploiting locality of information flow. Locality of reference is an established concept for program execution, which should therefore be exploitable within data-flow computations. Locality will be enhanced by the fact that functions are apt to reference their arguments repeatedly. Secondly, repeated global references to the same data will become localized by a *caching effect* which results from the implementation of such references. The latter will be further discussed in Section 12, dealing with the *apply* operator.

If computations which interact heavily with one another are allocated space in such a way that they are a shorter average distance apart in the nodes of the communication network, the overall time spent in information flow will be reduced. It is important to note that even if it is not possible to allocate space for a new computation in the address space of the same leaf, the correctness of the overall computation will be maintained, even though the speed of the computation may be degraded. This is a consequence of the uniformly accessible address space.

In designing a highly-parallel machine, one must be careful that costs involved in creating and communicating with new tasks do not outweigh the speed advantage gained from overlapped execution of these tasks. Consequently, our design prescribes that all computation local to a procedure body (i.e. exclusive of calls to other procedures) will usually be done within one processing unit. Hence, the global structure does not seek gains from parallelism on the level of, say, an arithmetic expression (although this could be done within the processing unit itself if desired), but rather from *inter-procedure concurrency*.

Another anticipated effect which will contribute to locality might be called the *seeding effect*. As shall be seen, when a task A in execution creates a second task B, the latter may be allocated its storage in any of the processing units in which there is sufficient space. Since B may cause the creation of other tasks C_1, C_2,...., C_n, locality is enhanced if the storage for the latter is allocated in processing units near to that of B in the tree. Hence, even if B is a long distance from A, thus incurring a major communication cost between the two, this cost may be balanced out by the lower costs of communicating between B and C_1, C_2,...., C_n. Hence, this seeding effect creates a tradeoff in resolving a choice of how far away a created task should be placed. It also demonstrates the possibility of a certain amount of *re-localization* in recovering from bad task-placement decisions by the system. For example, even if B is placed in a congested area, the storage from completing tasks near B can be reclaimed to provide more space for C_1, C_2,...., C_n.

6. INFORMATION FLOW

The characterization of information flow within the machine is very dependent on the conceptual level being considered. For example, at the *task level*, we are concerned with the flow of operands between tasks. In particular, our system permits *demand driven* computation at this level. In contrast, the machines of Dennis, Arvind and Gostelow, and Davis are all data-driven machines, in that an instruction never asks for data to be sent to it. Instead, it waits for data to be sent to it, and when all pieces of data are received, it initiates computation whose results are then sent to all other designated instructions. In the demand-driven scheme, a procedure may actively seek additional pieces of data after it has demanded and received some initial pieces of data. This topic will be further discussed in subsequent sections.

At the *communication network level*, we find the information flow separated into the flow of *tasks* (which are *invoke* instructions), *operands* (single data words), and *blocks* (multiple data words). All such pieces of information are accompanied by additional routing information in the form of destination addresses, etc. All information transmitted through the communication network is done by *packet switching* (or *store-and-forward*) as opposed to line switching. The latter type of switching is not used because of the potential congestion incurred by tying up long paths through the network.

A node of the communication network communicates to its parent through a traditional form of handshaking. However, for block transfers, a *burst mode* of communication is used in which the handshaking occurs only before and after the entire block has been transferred, thus drastically reducing the associated overhead.

7. MACHINE LANGUAGE

Our machine executes a compiled version of Lisp as its machine language. We avoid syntactic issues by using a *parallel program graph*, such as described in [Keller 77], instead of the conventional list representation of Lisp programs. For sake of definiteness, we refer to the graphical language as *Flow-Graph Lisp* (*FGL*). FGL allows us to clearly display the data flow between operators and thus potential concurrency within programs.

The equivalent of procedure calls, including recursive ones, is provided in FGL through graph *productions*, which specify how a programmer-defined operator (the *antecedent* of the production) is to be replaced by a program graph (the *consequent* of the production).

FGL also supports *lenient cons*, which allows the machine to exploit concurrency which it could not with conventional strict *cons* [Friedman and Wise 78]. For the current presentation, iteration is implemented by recursion, in the manner of [McCarthy 63]. This automatically gives the same concurrency-detection effect of "look-ahead" processors, which "unfold" iterations to achieve concurrency [Keller 75].

For sake of this presentation, let us suppose that data structures are *trees*, with the integers and *nil* as atoms. Boolean values may be implemented by interpreting *nil* as *false*, and any non-*nil* value as *true*. The program consists of a network of operators which are functions on trees. For simplicity, we do not discuss *input* of trees. Rather, we assume them to be resident at the beginning of the computation. Our trees are represented using an appropriate network of *cons* operators and atoms.

In summary, the program and all of its data are represented as one network in the machine, in a manner not too different from conventional representations of graphs in a linearly-addressable memory.

To cause a result to be printed, a *demand* is generated at some *print*

node in the network. This causes propagation of the demand to the operator feeding the *print*, which in turn eventually causes the value of that operator to be evaluated and printed.

Evaluation consists of a combination of transmutations to the graph and operations which produce new values from others. In this sense, we have a *reduction machine à la* [Berkling 75], executing a *reduction language à la* [Backus 73]. By using graphs rather than strings, we can avoid much of the *combinatorial explosion* which takes place in purely string-oriented machines.

Figures 2, 3, and 4 give examples of programs in FGL. In Figure 2, there is a *main program* M. M calls a *recursive procedure* g, the graph of which is presented in Figure 3. In each figure we give the graph representation and the corresponding "code block" representation (see Section 8). The parenthetic labels on the graph indicate the correspondence between the two. Intuitively, g(n) "computes" the infinite sequence

n n+1 n+2 n+3 ....

In the context of the main program, the value printed is the third element (*caddr*) of the sequence with n = 0.

A second program, which sums a tree of integers, is shown in Figure 4. This example uses a *strict* operator, *add*, to cause the creation of instances of operators which can be evaluated concurrently. Figure 5 shows a possible snapshot of the program during its application to a specific tree.

In the next sections we describe, in more detail, program storage, task execution, typical operators, graph expansion via the special *invoke* operator, and *forward chaining*, which is a key idea in implementing *lenient cons* and our particular form of procedures. We do not discuss storage reclamation here, as it is an issue still under investigation.

8. PROGRAM EXECUTION

All storage is allocated in *blocks*. Blocks make storage management more efficient, and are consistent with trying to keep the *locality* of a computation contained with one processing unit. A block is either a *data block* or a *code block*. The words of a data block are initially code and literals. The former gradually get changed to data during execution. A code block is copied as the source of initial code to be stored in a newly allocated data block. The contents of a code block form a linear representation of an FGL program graph.

The copying of code blocks may be contrasted with approaches such as that in [Patil 67], which interpret a pure code block without copying. The approach taken here is more effective in keeping references local to a processing unit. It also reduces the amount of word fetching required during actual task processing.

The words in a data block correspond roughly to data values which may eventually appear on the output arcs of operator nodes in the program graph. Initially however, instead of containing data, a word contains the *instruction code* representation of the corresponding operator, along with the local addresses of words corresponding to its input arcs, i.e. the sources of its operands. We assume here for simplicity that each operator has only one output arc, although such arcs may fan out as necessary.

In addition to specifying the input arcs of its operands, an instruction may be accompanied by *notifiers*, which are addresses of operators which have this operator's output arc as one of their input arcs. These could conceivably be set dynamically, but in this presentation we have elected to have them set in the initial code. Again, Figures 2, 3, and 4 give examples of code blocks corresponding to program graphs. Further information on interpreting these blocks is given in subsequent sections.

By keeping data blocks reasonably small, say 256 words, and by using only addresses relative to the start of the block in the code, the operation code and necessary set of operand and notifier addresses can be accommodated within a reasonable word size, say 48 bits. For references across blocks, which therefore involve global addresses, we provide some special operators, to be described subsequently. By dividing the physical memory into blocks and allocating on block boundaries only, a *paging effect*, which simplifies storage management, is readily obtained.

9. TASK EVALUATION

The loosely-coupled aspect of task evaluation is achievable through a *task list* organization, which allows many processors to partake in the evaluation of *tasks*, i.e. particular instances of operators with their associated data. The task list is decomposed into two separate lists which may be served independently. These are:

demand list: contains addresses of operators for which evaluation is to be attempted.

result list: contains addresses of operators, along with their corresponding values after evaluation.

At this stage of development, the recommended priority of service is result first, then demand. The reasoning here is that result values generally enable successful evaluation of tasks, while demand generally creates more tasks. These lists are further divided and distributed to individual processing units by the communication network, which takes into account the current processor load distribution. Only *invoke* instructions will be considered for distribution, for it is only these which might profitably be executed in another processing unit, due to the communication cost incurred in getting them there. Hence, the *invoke list* is a sub-list of the demand list, containing only *invoke* instructions.

Figures 6 through 11 show the organization of the task evaluation mechanism. The flow diagrams are to be interpreted in an informal sense, and are less akin to conventional flowcharts than they are indicative of *data flow*, with *tasks* as data.

The following brief narrative will aid in the understanding of the flow diagrams. Initially, the address of the word which will produce the "main result" is put on the *demand list*. The word itself is then fetched. It is evaluated, if possible. If not, then demand is propagated to its arguments by placing their addresses on the demand list.

Once evaluated, a result value replaces the coded operator as *ready* data. Via the result list, any notifiable operators awaiting this result as an argument are then notified by putting them on the demand list to be retried. We notice that all demanded operators remain accessible until they become ready as data, either through:

(1) being on the demand list,

or (2) being referenced by a notifier of an accessible operator,

or (3) being referenced by the "forwarding address" of an accessible operator.

Forms of evaluation other than pure demand evaluation can thus be supported by judicious setting of "*d*-bits" and advanced placement on the demand list.

10. WORD FORMAT

A word in a data block may begin as a *code word* and later be changed to a *datum* as the computation proceeds, corresponding to the evaluation of the operator represented by that code word. The *ready bit* (*r bit*) in each case is set when the word does contain a datum. It may be set initially in some words, to provide initialized literals.

A datum can either be an *atom*, in which case it contains a literal value, or it can be a *pair pointer*. In the latter case, it is the global address of a pair. A *pair* consists of two consecutive words within some block, each of which is either a datum or a *forward* operator. The purpose of the latter will be described subsequently.

There are several other formats for data which extend the above, such as representing lists in contiguous space, chains of pointers, etc., as used in [Bawden, *et al.* 77]. These will not be discussed here for brevity.

All global addresses are represented as

B.R

where B is the base address of a data block and R the local address of a word within the block. The advantage of this scheme is that once the word in question has been referenced, the processor will usually need access to other words in the block and can gain it using only their local addresses.

The following fields will always be present in a code word:

d bit: set to indicate that its ultimate data value has been *demanded*

op : *operation* code

The following fields may or may not be present, depending on the nature of the particular operation code:

as : local addresses of *arguments* to the operator

ns : *notifiers*, i.e. local addresses of *notifiee* operators

→B.R : where B.R is a global address, which is either:

a *forwarding address*, which is used with a *forward* operator, or

a *fetch address*, which is used with a *fetch* operator, or

a *pointer* to a code block (in which case R = 0), which is used with an *invoke* operator.

The presence of the demand bit in a code word allows support of a *demand driven evaluation strategy*. In this strategy, no operator is evaluated unless it produces some value known to be essential to the computation. Aside from the obvious potential efficiency gain, another advantage of this approach is that it provides a natural means of deciding whether and when to trigger the invocation of a defined function, which requires the allocation of a storage block.

The use of bits to direct the processor to interpret a given word as data, instruction, etc. exemplifies the "tagged architecture" approach [Feustel 73]. Adopting this approach allows us to keep open all of its attendant options as the design progresses.

11. REPRESENTATIVE OPERATORS

The repertoire of operators includes the Lisp operators *car*, *cdr*, *cons*, *atom*, *eq*, *if-then-else (cond)*, *etc.* Of these, all but the first three are called *ordinary*, as they operate purely within the data block. The first three are called *special*, because they can cause data transfer between blocks.

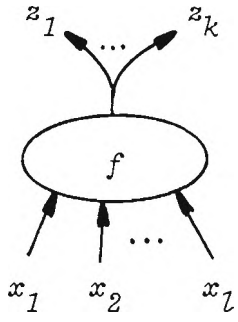
In contrast to conventional Lisp, we have elected to make *cons* a *lenient* operator. That is, it has a "result" even if one of its arguments has not yet been computed. This can be argued to increase the asynchrony of a computation and hence improve the utilization of a parallel processing system on which it may be run, *cf.* [Friedman and Wise 76].

A consequence of the lenience of *cons* is that, in our implementation, *cons* is not really an operator at all, but rather just a pair of data, namely, its arguments.

Some other special operators, which do not appear in the program graph, are used to effect the necessary transfers of data between procedures, and other housekeeping operations. These are *ident*, *forward*, *fetch*, *locptr*, and *invoke*.

The operators *ident*, *forward*, and *fetch* all have the nature of *identity functions*. The distinction is as follows: *ident* has a local argument and local notifiers. It is used mainly for increased fan-out when there are more notifiers for a word than can fit in a single word; *fetch* has a global argument and one or more local notifiers; *forward* has one local argument and one global *forwarding address*. The latter is set when a demand is issued to the corresponding *fetch*. All *cons* pairs compile as two consecutive *forward* operators, or literals. The operator *locptr* is used to generate global pointers to *cons* pairs.

The following discussion describes the compilation of an *invoke*:



where f is a programmer-defined symbol

compiles as:

invoke →f nz_1 ... nz_k
forward ax_1 →?
forward ax_2 →?
.
.
.
forward ax_l →?

where →f is the address of f's code block, the ax_i are local arguments, the nz_i are local notifiees, and the ?'s are set when the *forward*s are demanded.

The data block corresponding to f begins with:

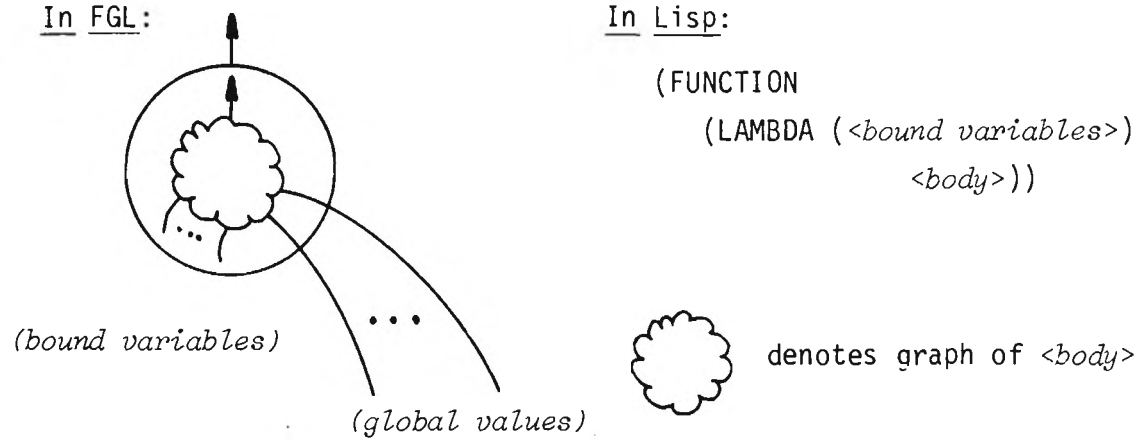
forward au →χ
fetch →(χ+1) ny_1 ...
fetch →(χ+2) ny_2 ...
.
.
.
fetch →(χ+l) ny_l ...

where χ is the address of the *invoke*, u is the local word which will contain the result to be delivered by the *invoke*, and ny_i ... are the notifiers of the i-th parameter of f. Following creation of the data block, demand propagates to the *forward* in the data block for f.

12. FUNCTION CLOSURES AND THE OPERATOR *apply*.

An important aspect of Lisp programming is the manipulation of functions as data values. While we do not envision supporting run-time *creation* of function *definitions*, we do accommodate the formation and manipulation of function *closures* (records combining compiled code pointers with environments for their ultimate application; i.e., FUNARGs). This will permit not only the programming of *functionals* (function-valued functions) on our machine, but also provides a form of shared values, thereby relieving the need to exhaustively parameterize functions.

We assume that our programs are *block compiled*. That is, the program consists of a set of symbolically named function definitions that are compiled as a group. Within these "top-level" function definitions, there may be some number of nested function definitions of the following form:

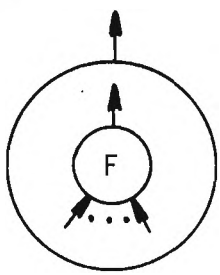


Such forms create *closure* values at run-time. Each combines the entry point for the nested function's compiled code with an *environment pointer* which references the currently executing activation of the immediately surrounding function definition. Thus global (i.e. "free", or non-local) variable occurrences within the nested function are bound *statically* to

refer to the matching declaration (i.e. parameter) binding at the place of the closure's creation.

For completeness, we include:

In FGL:



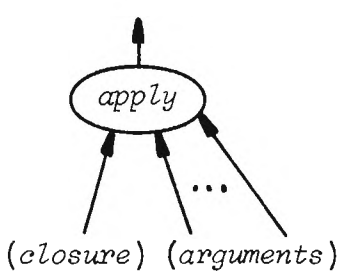
In Lisp:

```
(FUNCTION F)
```

where F is the symbolic name of a function. This makes the semantics of function application more uniform, and syntactically distinguishes between the function F and any parameter F that may be accessible. Note, however, that the environment pointer in such a closure is superfluous, since a named function may not contain any occurrences of variables global to it.

A closure value may be passed as a function argument, returned as a function value, conditionally selected, etc. until ultimately it is applied via the operator *apply*, akin to the APPLY function of Lisp:

In FGL:

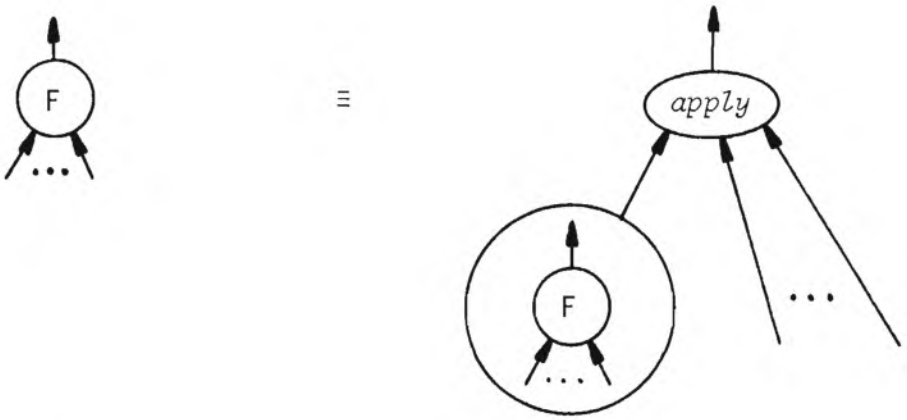


In Lisp:

```
(APPLY <closure-form>
       <arg1-form>
         .
         .
         .
       <argk-form>)
```

Observe that all function calls in our source language could be expressed in APPLY notation through the following transformation:

In FGL:



In Lisp:

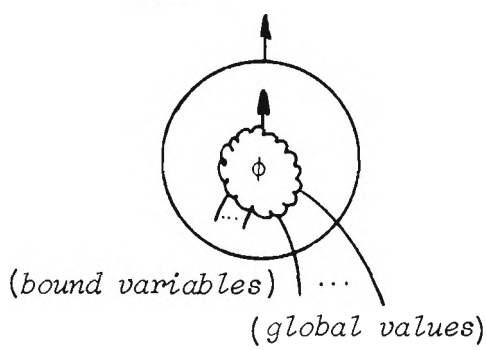
$$(\text{F}\ \alpha_1\ \ldots\ \alpha_k) \quad \equiv \quad (\text{APPLY (FUNCTION F)}\ \alpha_1\ \ldots\ \alpha_k)$$

However, we retain the option of the direct function call notation (and the *invoke* opcode supporting it) for expressive convenience and run-time efficiency.

These constructs are compiled as follows (see Fig. 12 for examples):

Construct 1: Function closures.

In FGL:



In Lisp:

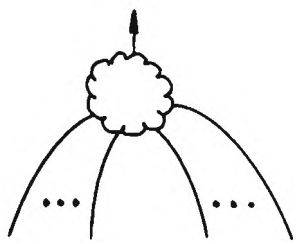
(FUNCTION ϕ)

We use the opcode *locptr* to generate a full-address (i.e. "global") pointer to a *cons* pair representing the closure. The *car* of the pair is the

keyword atom FUNARG, while the *cdr* is a pseudo-opcode *dummy* with a code pointer to ϕ as its argument. Thus the *car* of a closure may be computationally inspected at run-time, but since *dummy* causes a run-time error if executed, the *cdr* of the closure is inspectable only by *apply*. Note that the global pointer S.β to the closure as built by *loadptr* contains the closure's environment pointer directly in S.

Construct 2: Nested functions.

In FGL:



(*bound variables*) (*global values*)

In Lisp:

(LAMBDA (<*bound variables*>) <*body*>)

Each function definition is compiled into a separate code block to minimize code copying at function application time. (Note that if nested functions were compiled "in-line", their code would still need to be copied when applied, since several applications of that particular closure value may occur.) Within each function's code, special "pseudo-parameter" *fetch* opcodes are compiled for each variable accessed globally from within its definition. Observe that such *fetch*es are compiled even for global variables accessed only at deeper nested function levels.

Any global variable occurrence is thus connected at run-time through a sequence of *fetch* opcodes, one per level of textual function nesting, from its containing activation record to its binding as a *bona fide* parameter at some outer level. The S.β pointers of the global *fetch*es are bound in two stages: the β is fixed at compile time (with complete security), and the S is fixed

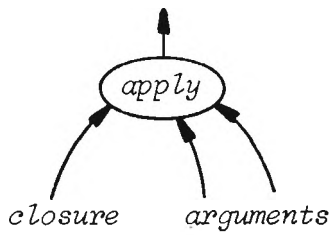
at application time to be the closure's environment pointer S.

Thus, in the same sense that the activation record's *dynamic* (i.e. calling) link is redundantly represented in each parameter *fetch*, its *static* link is redundantly represented in each global *fetch*. The *fetch* opcode offers sufficient space for such full addresses, and the design provides uniform *fetch* processing in both cases with less memory contention (as might arise if the static and dynamic links were put into a single header word in the activation record).

An alternative accessing scheme for globals would be to replace this "bucket brigade" approach and provide direct *fetch* linkage from occurrence to binding levels. Although such a scheme might offer faster access in certain cases, we consider it to be less desirable for two reasons. First, the compiled code would need to be adapted to contain two-dimensional addresses (i.e. [*static level*, *offset*], as is customary in Algol-like language implementation), with the added application time set-up activity. Secondly, a potentially valuable *caching* effect would be lost along global *fetch* sequences. Given our concern for exploiting locality on this machine, we feel that the latter concern will be economically dominant.

Construct 3: Function applications.

In FGL:



In Lisp:

(APPLY <*closure-form*>
<*arg*$_1$*-form*>
.
.
.
<*arg*$_k$*-form*>)

The *apply* operator is compiled in a manner similar to that for *invoke*, but with the *closure* being an operator argument (as opposed to the

arguments, which are compiled using *forward*s as per *invoke*).
The actions taken by the *apply* opcode are viewed as a slight extension of the *invoke* opcode, with the added activities of global *fetch* set-up and argument count checking. When demand reaches an *apply* operator, it propagates immediately to the *apply*'s first argument. Upon receipt of the necessary closure value for this argument, the *apply* task becomes an *invoke* task and is moved to the *invoke* list.

13. *forward* CHAINING

The narrative in Section 9 does not discuss special attention paid to various operators, e.g. *forward*. The handling of such operators is the essence of both the procedure linkage mechanism and the successful handling of lenient *cons*.

When an operator is evaluated, it is replaced with a value. At this time, the presence of any notifiers is noted and the corresponding operators are put on the demand list. These operators can then access the data as an operand.

No use is to be made of the argument part of the contents of operators over-written by *forward*. Instead, a special *forward chaining* technique is required for consistent handling of lenient *cons*. If the operator being replaced is a *forward*, the data also replaces the contents of a *forwarding address* which may be present. This process is repeated, until an operator containing no forwarding address is encountered. The need for this technique can be seen by the following argument:

Figure 11a shows parts of three data blocks as part of a state. Notice that X_1 and X_2 can both potentially request the same value, namely the value of U, which is not yet ready (nor demanded). When the first demand on Z is generated, as indicated in Figure 11a, the forwarding address in Z is set to X_1 and U is demanded.

Suppose meanwhile that demand is generated on X_2, which in turn results in a second demand on Z. Since a forwarding address has already been stored in Z, there is insufficient room for a second. (Even if two could be stored, there might be three demands generated, etc.). Since we know that X_2 is to receive the *result* of Z, we store *forward* $\rightarrow X_2$ in Z, as in Figure 11c. When U is finally notified, any data stored over a *forward* will be stored over the contents of the

word specified by its forwarding address, according to the distribute/notify phase of the evaluation algorithm.

Although we used *car* to motivate the above example, we mention that similar treatment is given to *cdr* and *fetch* (when used for global value linkage).

14. PROCESSOR ARCHITECTURE

We do not go into great detail here on the organization of individual general processing units. As described in Section 9, each unit selects tasks from its demand list. While on this list, a task is represented by its address in memory. This word is fetched and if not presently ready as data, an attempt is made to evaluate it. For *ordinary* tasks, this normally entails reference to one or more additional words in the memory; hence a fetch of these words occurs. Since each of them might reside in the physical memory of any processing unit, fetching may involve transmission of words through the communication network. In order that the processor need not be idle while such a fetch is taking place, we provide for buffering a set of such tasks while their operands are being assembled. We call such a buffer a *staging area*. It is conceptually similar to a conventional *pipeline*, except that order of task execution is unimportant, all essential ordering being explicit in the program graph. The size of the staging area is chosen to maintain reasonably good utilization of the function units within the processing unit, which carry out the actual operations once the task leaves the staging area. Of course, each function unit could itself be pipelined, depending on economic advantages which would accrue due to a particular application load. Design of such a staging area is fairly routine and therefore will not be further discussed here.

15. LOAD BALANCING

Load balancing occurs through the redistribution of tasks from the invoke list of one processing unit to that of another. This is a separate, but topologically comptabile, function of the communication network from the routing of operand data.

By the *load* at a processing unit, we mean the number of tasks on the segment of the invoke list at that unit. In a similar manner, we can define the *load* at any node of the communication network to be the sum of the loads at its leaves, divided by the number of its leaves as a normalizing factor.

Again, to simplify the explanation, we are assuming that the communication network is a binary tree. Each node of the communication network maintains lower and upper limits, L and U, on the loads of its immediate descendants. If the load of one is above U and that of the other below L, it attempts to shift tasks from the invoke list of the overloaded descendant to that of the underloaded one. If loads of both its descendants are above U, this will be communicated to its parent (if any), so that the latter may try to shift some of the load to one of *its* descendants having load less than L. In this way, the balancing function is distributed throughout the communication network, with each node thereof applying the same balancing strategy.

The effectiveness of the balancing scheme relies on the loosely-coupled aspect of the system. That is, no task is bound to a particular processor until storage is allocated for it.

16. COMPARISONS WITH RELATED MACHINES

It is easiest to understand the relation between the machine architecture presented here and the architecture of the data-flow computer proposed in [Dennis and Misunas 74] by folding the latter through the center of its instruction cells and functional units in such a way that the arbitration network overlaps the distribution network. Our general processing units then play the role of the instuction cell blocks, and our communication network performs the function of both arbitration and distribution networks. Furthermore, our architecture may offer improved performance because data would not often have to travel as far to get from a source cell to a destination cell.

As in the machine proposed in [Arvind and Gostelow 77], the machine proposed here uses micro-computers to do the processing. However, we feel that the communication network used in our machine is superior to the one in that machine. The communication bus structure of the former machine may cause intolerable delays in transmitting information from one processing unit to another, a fact that may prove to be a great impediment to the success of the machine.

The DDM-1 [Davis 78] is a very different kind of machine than the one proposed here. Its hierarchical structure seems to impose certain constraints on the creation of new computations and on the flow of information in the machine. For example, when a processing element creates a task, the latter must be placed either in the space of the processor carrying out the application or in the space of a subordinate processor, even if the subordinates are crowded for space and the machine has other processors which have plenty of free space. This problem does

not occur in our machine, due to the construction of the communication network, the uniformity of the address space, and our notion of load balancing.

Some tree-structured reduction language machines that have been proposed are fundamentally different in their operation when compared with the machine presented here. In these machines, the expressions that need to be evaluated are mapped directly onto the physical tree of the machine. In our machine, such expressions would not be mapped onto the communication tree; instead they would be mapped via parallel program graphs into the address space of the machine, and would reside in the memory space of one or more processing units of the machine.

A common feature of all of the above architectures is that they are data-driven rather than demand-driven, as ours is. One might be led to think that the latter presents some additional overhead. However, closer examination of the other architectures may reveal that some form of ready-acknowledge signalling is taking place when it comes to transmission of data via storage words. This is, in fact, a special case of demand-driven computation, in which the demand for an operand is equated with readiness of its recipient. We exploit the flexibility of the general case, to obtain advantages in deciding when to invoke procedures. It is also clear that the demand-driven feature is a necessity in supporting lenient *cons*. One the other hand, it is also clear that demand-driven computation can be *engineered* on the other architectures by treating demands as data, but this seems to be cumbersome.

Although at the physical level the Cm* computer [Swan, *et al.* 77] may appear similar to our machine, the two are quite different on account of their underlying mechanism of program execution. In Cm*, parallel

processing is based on the concept of interacting sequential processes that run on conventional processors (PDP-11), while our machine embodies an evaluation scheme for the FGL language and is capable of directly evaluating data-flow graphs and applicative expressions. Our evaluation scheme, language, and overall organization have been developed in an integrated fashion as parts of one functioning system.

17. CONCLUSIONS AND FUTURE RESEARCH

We have stated our feeling that machine architectures should be developed with greater attention paid to ultimate programmability. As an example, we discussed principles for a loosely-coupled architecture and the use of Lisp as a language well-suited for such a machine. We sketched in some detail the internal representation of programs in our machine and the execution of programs on it.

Our implementation seems to be the first detailed one presented for Lisp programs on a parallel machine. An implementation has been described qualitatively in [Friedman and Wise 78]. However, their work relates mainly to the issues associated with *colonel* versus *sergeant* tasks, the latter being distinguished from the former as tasks whose evaluation may never be actually required, but which provide a potentially useful way of employing otherwise idle processors. In contrast, all tasks in the machine described here are of the colonel variety, whose existence may be traced to certain *strict* operators, such as *add* in the tree sum example. Hence such issues have not been of immediate concern here. On the other hand, subtle details, such as the need for *forward chaining* have been discovered in the course of designing our evaluator. How such subtleties interact with an implementation which does support sergeant tasks remains a topic for future investigation.

The ideas presented here were derived after considering many possible alternatives. It is, of course, possible that we may elect to return to one or more of these alternatives after more experience in programming the machine has been gained. A simulator for the evaluation model has been written in Pascal to assist in such a venture.

Many important details remain to be investigated. These include not only the necessary support for the language described here in terms of storage reclamation and scheduling, but extension of the language to allow other features as well. We are currently contemplating how to best introduce a distributed heap for more efficient long-term data storage. We must decide how to deal with other features of Lisp, such as *prog*, upon which many programmers have learned to rely. A related issue is whether *indeterminate* computations should be supported, as there are some indications that they permit efficiency gains not otherwise achievable [Keller 78]. The usefulness of applicative programs in allowing graceful backup when a processing unit fails also remains to be explored. Thus many issues, at levels from detailed processor construction to more fundamental language problems, await us.

ACKNOWLEDGEMENTS

Comments by Al Davis, Milos Ercegovac, and Mark Franklin, as well as encouragement from Jack Dennis, are appreciated.

The authors express their thanks to Kathy Burgi, Jodie Doyle, Karen Evans, Lujuana Fornelius, and Mary Ann Kleinert for their assistance in preparing the manuscript.

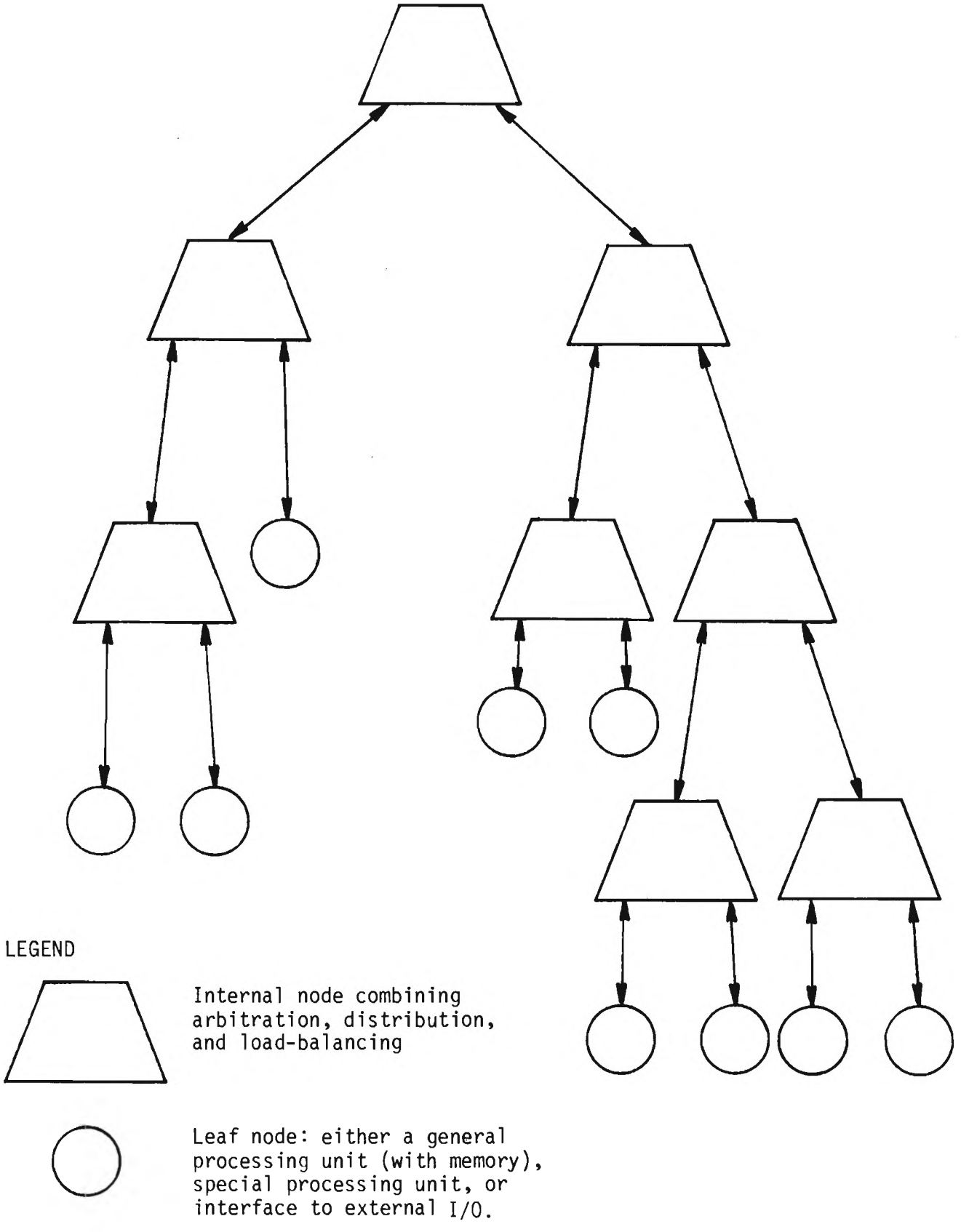


Figure 1 Form of the physical architecture of the loosely-coupled parallel processor.

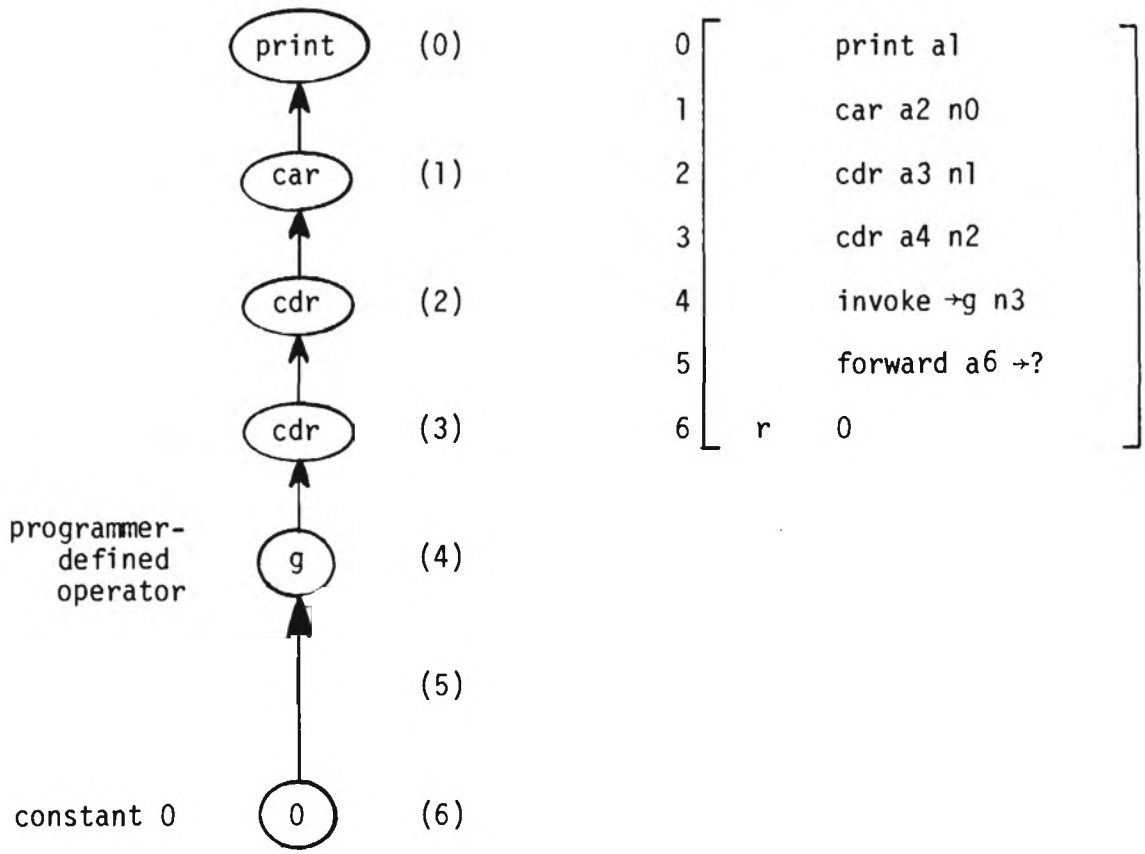


Figure 2 Graph representation and initial data block for sample *main program* M.

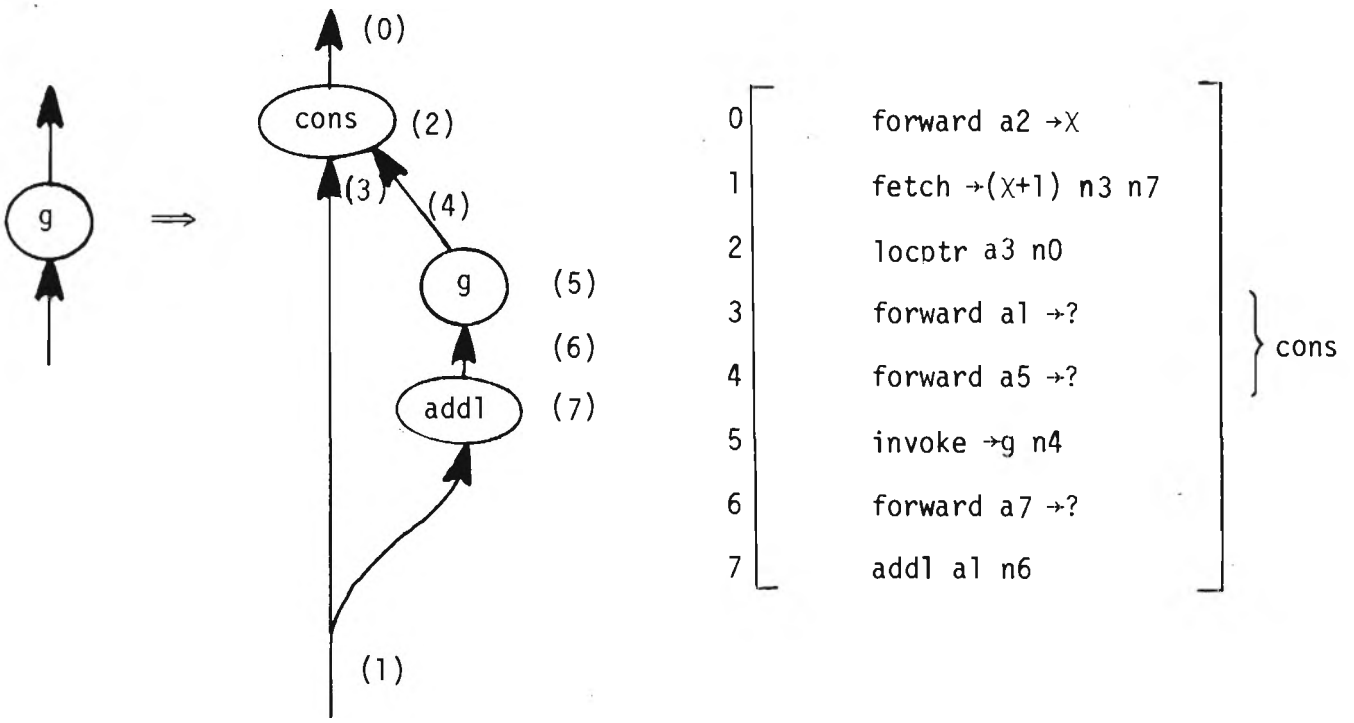
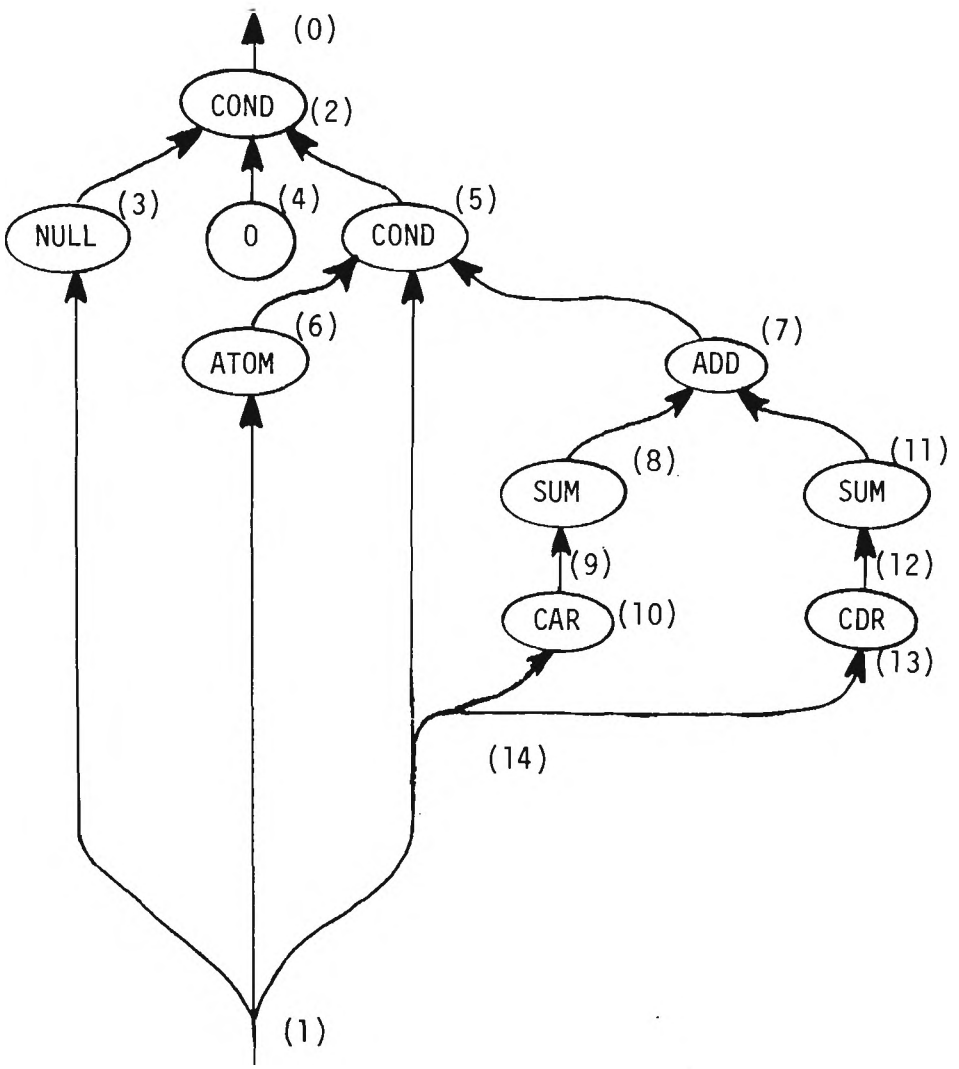


Figure 3 Graph representation and code block representation of the consequent of a production. X is the global address of the *invoke* operator which creates the corresponding data block. ? indicates pointer fields which are set on demand of this word. *locptr* is an operator which generates the global address of the word it references.

```
(DE SUM (TREE) (COND
     ((NULL TREE) 0)
     ((ATOM TREE) TREE)
     (T (ADD (SUM (CAR TREE)) (SUM (CDR TREE))))))
```




```
0  forward a2 →χ
1  fetch →(χ+1) n3 n6 n14
2  cond a3 a4 a5 n0
3  null a1 n2
4  r 0
5  cond a6 a14 a7 n2
6  atom a1 n5
7  add a8 a11 n5
8  invoke →sum n7
9  forward a10 →?
10 car a14 n9
11 invoke →sum n7
12 forward a13 →?
13 cdr a14 n12
14 ident a1 n5 n10 n13
```

Figure 4 Tree summation example: Lisp code; consequent of production defining SUM; compiled code.

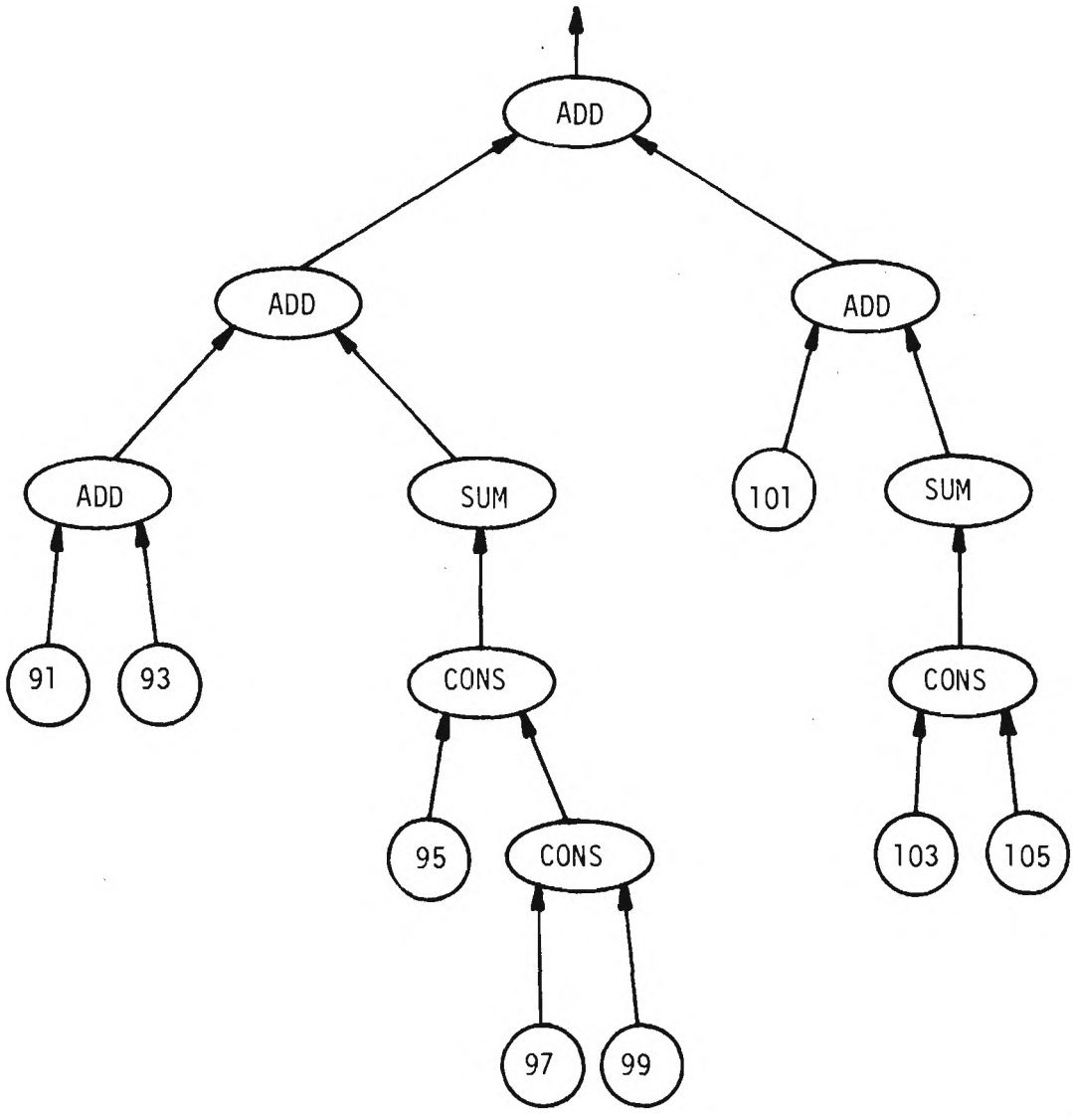


Figure 5 One possible snapshot of the program of Figure 4 during its computation on a tree.

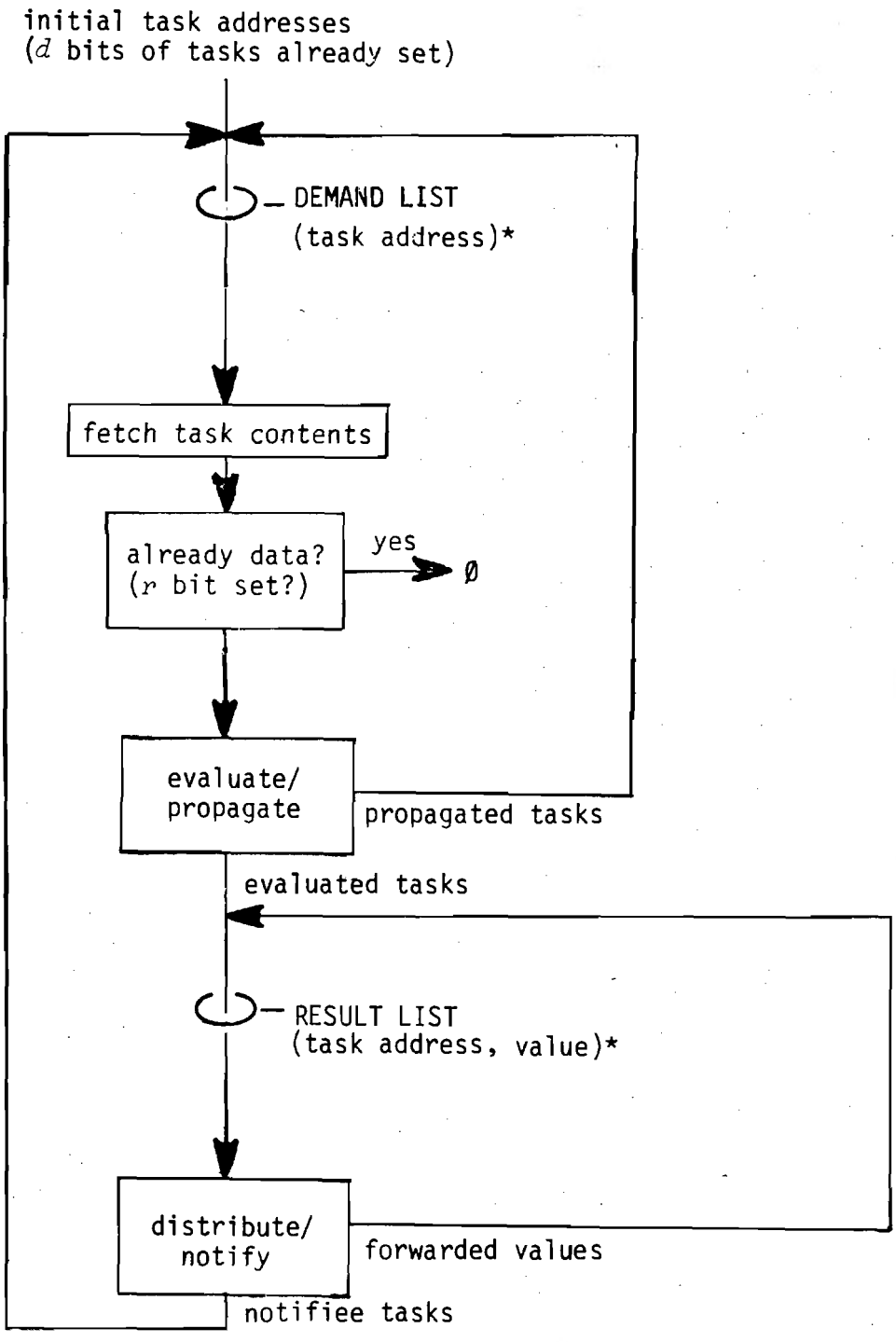
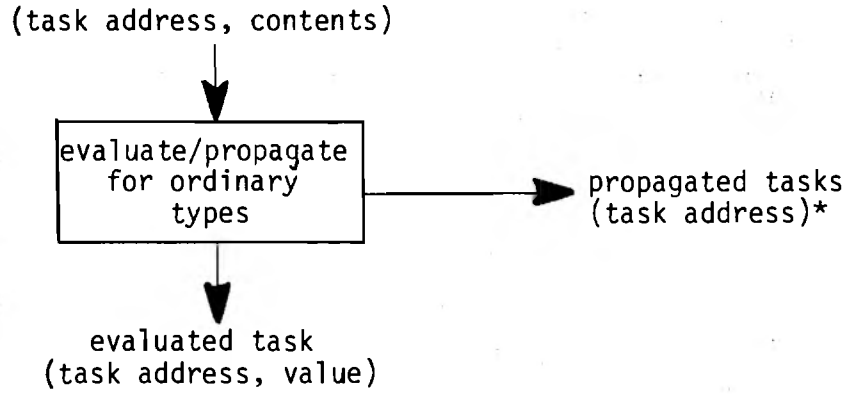


Figure 6 Overall task processing flow. Asterisk denotes *sequence of*. The evaluate/propagate box for different task types is expanded in Figures 7, 8 and 10. The distribute/notify box is expanded in Figure 9.



expands into:

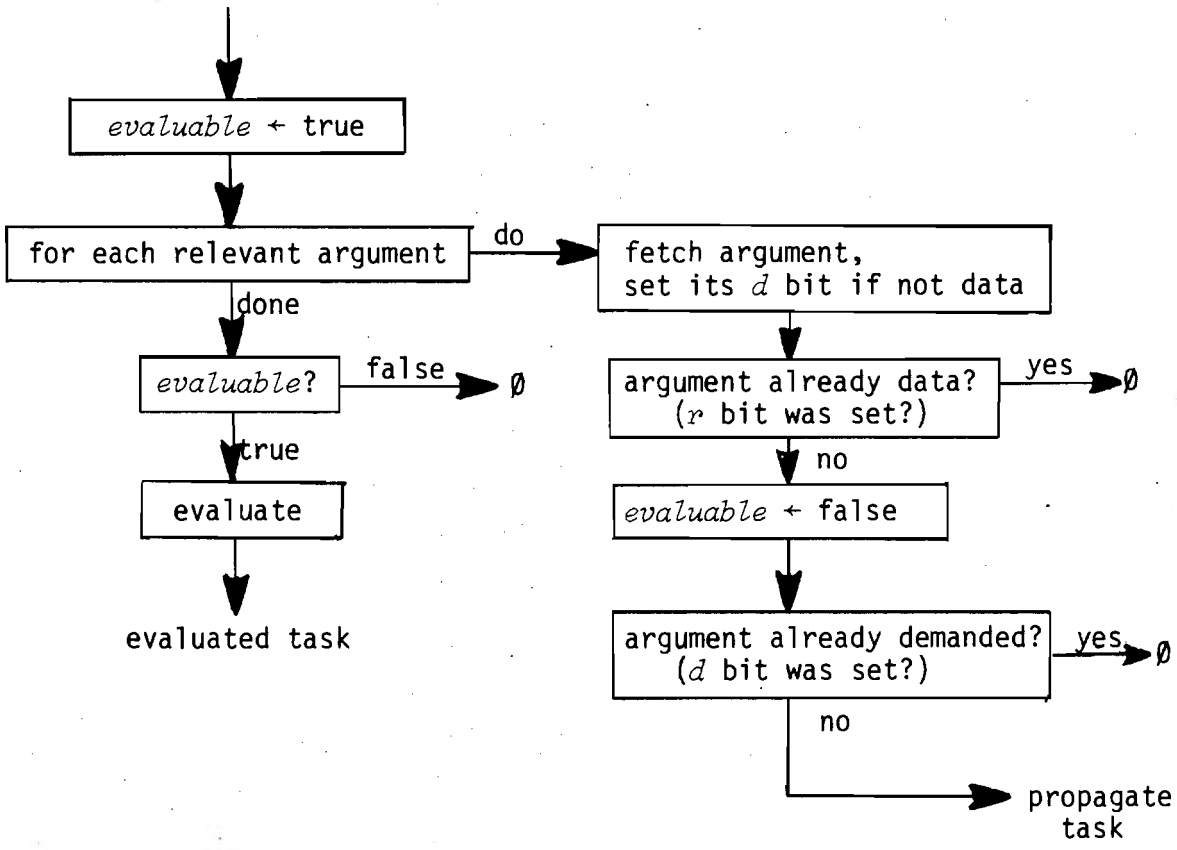
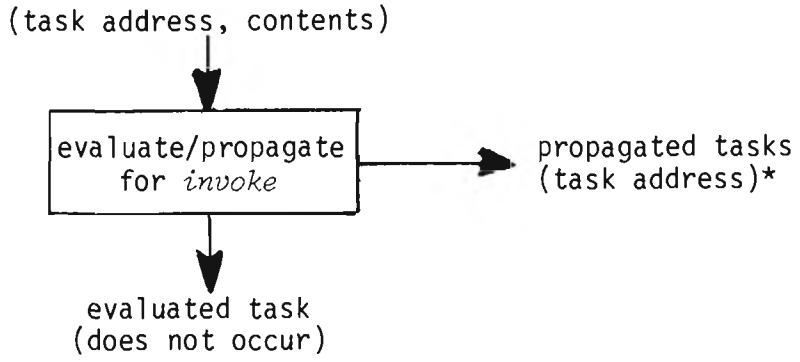


Figure 7 Evaluate/propagate for ordinary task type.



expands into:

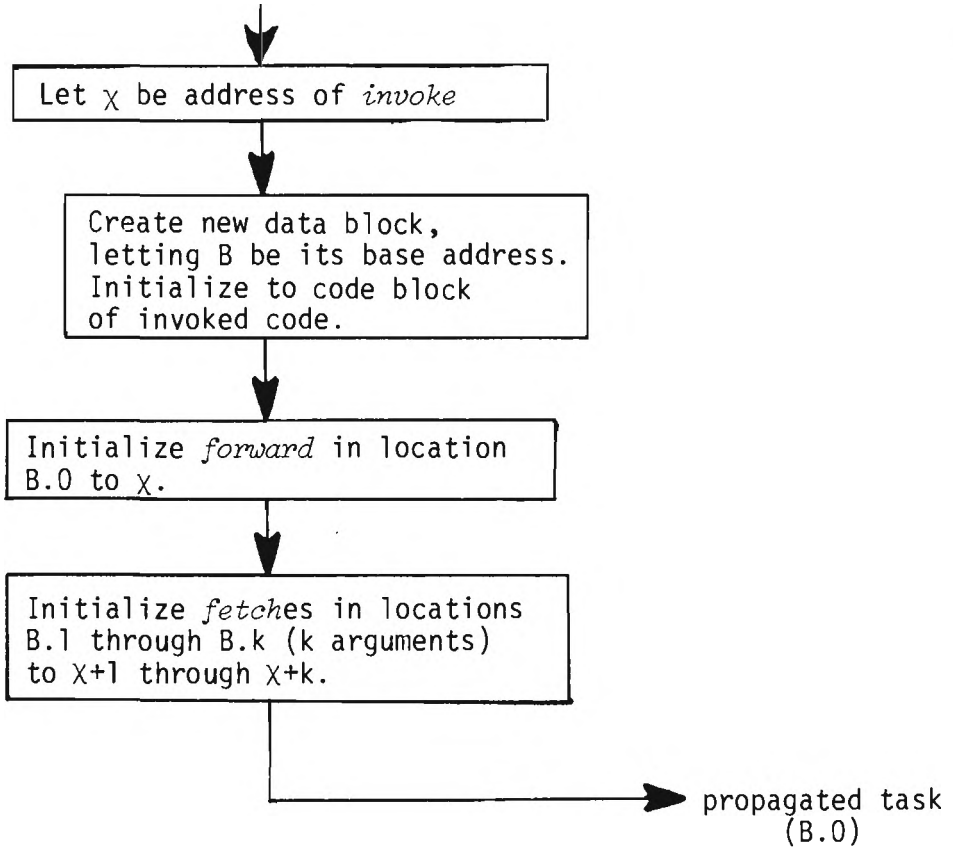
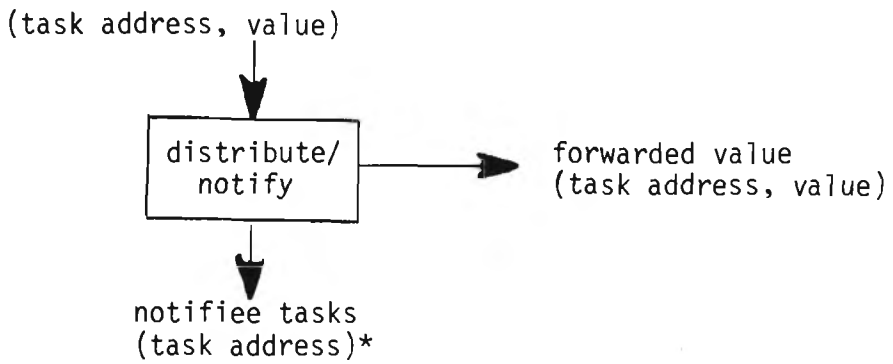


Figure 8 Evaluate/propagate for *invoke* task type.



expands into:

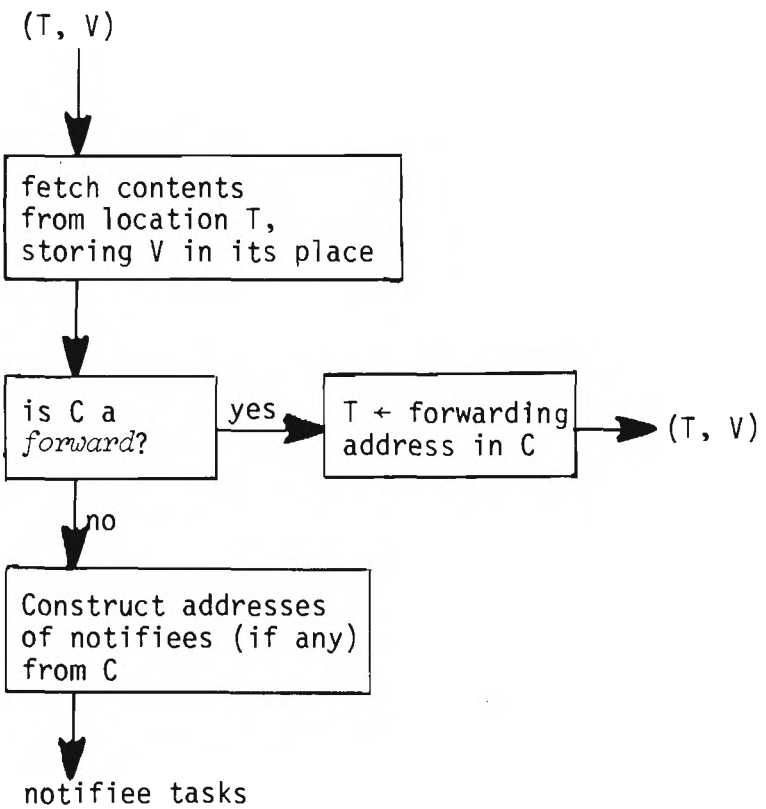


Figure 9 Distribute/notify processing.

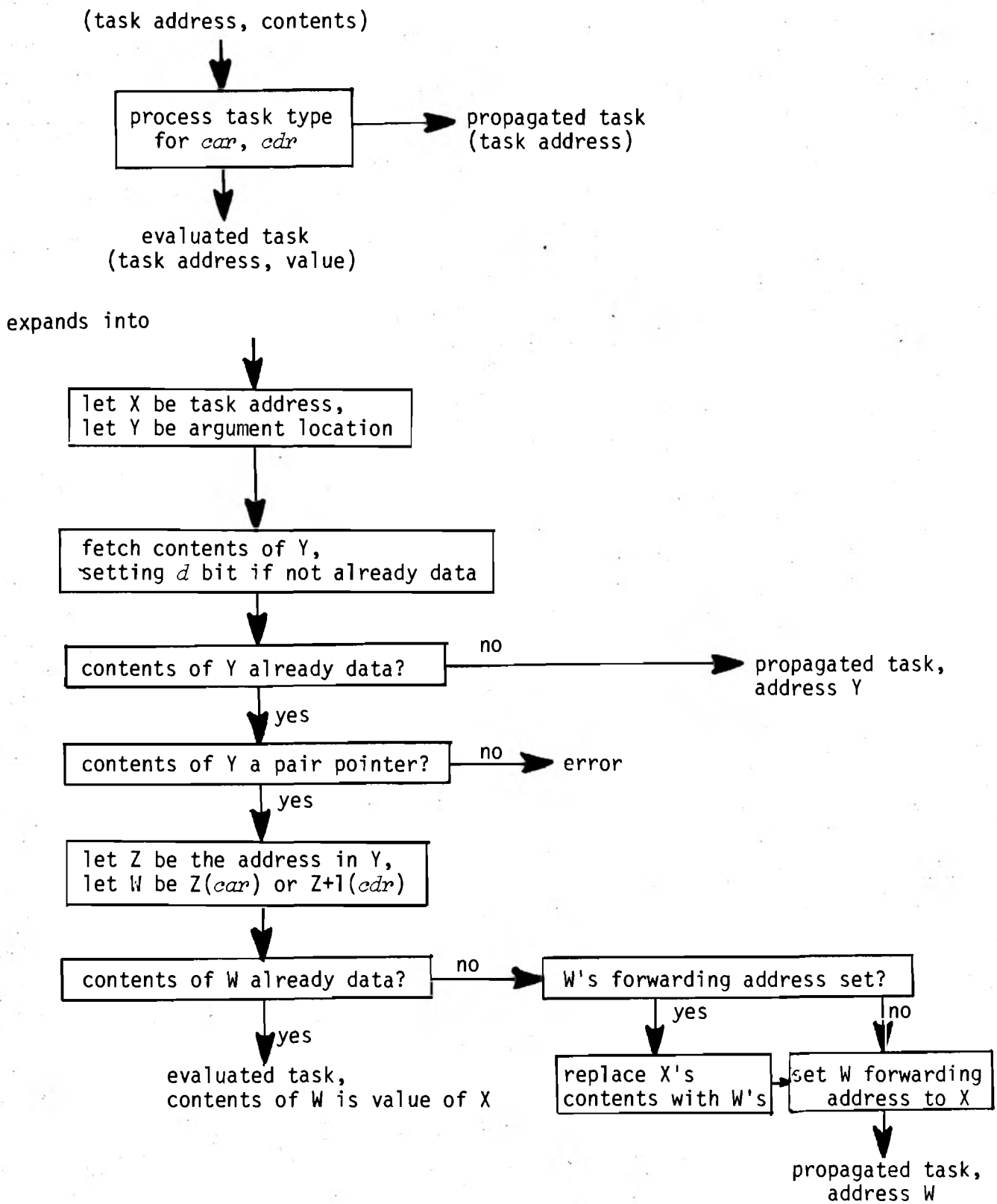


Figure 10 Evaluate/propagate for *car*, *cdr* task types.

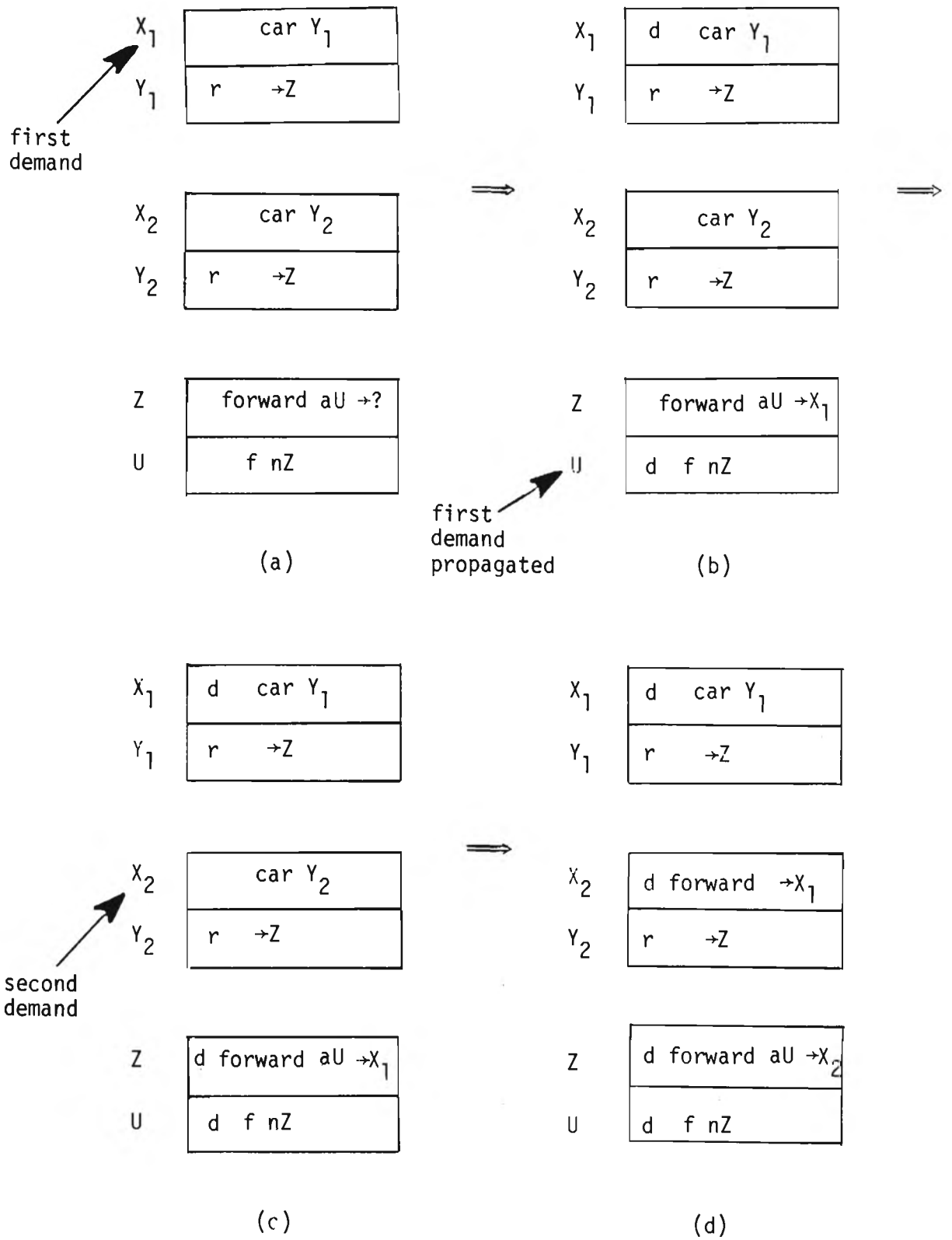
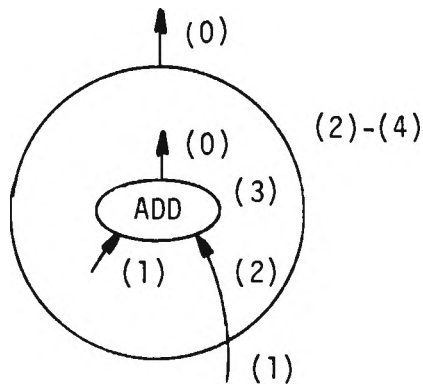


Figure 11 Illustration of *forward* chaining. (r and d denote *ready* and *demand* bits, respectively.)

Lisp code: (DE ADDK (K) (FUNCTION (LAMBDA (J) (ADD J K))))

FGL code:



Compiled code:

```
ADDK:  0  [  forward a2 →χ   ]
       1  |  fetch →(χ+1)     |
       2  |  locptr a3 n0     |
       3  |r 'FUNARG'         |
       4  |  dummy →α         ]

α:     0  [  forward a3 →χ   ]
       1  |  fetch →(χ+1) n3  |
       2  |  fetch →(ξ+1) n3  |
       3  |  add a1 a2 n0     ]
```

Figure 12 Simple example of function closures: "Currying" the operator *add* to have a bound second argument. (χ denotes the dynamic link, and ξ denotes the static link, both bound at *invoke* time.)

REFERENCES

[Arden and Berenbaum 75] B. W. Arden and A. D. Berenbaum. A multi-microprocessor computer system architecture. Operating systems review, 9, 6, 114-121 (Nov. 1975).

[Arvind and Gostelow 77] Arvind and K. P. Gostelow. A computer capable of exchanging processors for time. Proc IFIP '77, 849-853 (1977).

[Backus 73] J. Backus. Programming language semantics and closed applicative languages. Proc. ACM Symp. on Principles of Programming Languages (1973), 71-86.

[Bawden, *et al.* 77] A. Bawden *et al.* Lisp machine progress report. MIT AI Memo No. 444 (August 1977).

[Berkling 75] K. J. Berkling. Reduction languages for reduction machines. Second Annual Meeting of Computer Architecture (1975), 133-138.

[Davis 78] A. L. Davis. The architecture and system method of DDM-1: A recursively-structured data driven machine. Proceedings of the Fifth Annual Symposium on Computer Architecture (1978).

[Dennis and Misunas 74] J. B. Dennis and D. P. Misunas. A preliminary architecture for a basic data flow processor. Proc. 2nd Annual Symposium on Computer Architecture, 126-132 (Dec. 1974).

[Fateman 73] R. J. Fateman. Reply to an editorial. ACM SIGSAM Bulletin, No. 25, 9-11 (March 1973).

[Feustel 73] E. A. Feustel. On the advantages of tagged architecture. IEEE Trans. on computers, C-22, 7, 644-656 (July 1973).

[Friedman and Wise 76] D. P. Friedman and D. S. Wise. CONS should not evaluate its arguments. in Michaelson and Milner (eds.), Automata, Languages, and Programming, 257-284, Edinburgh University Press (1976).

[Friedman and Wise 78] D. P. Friedman and D. S. Wise. Aspects of applicative programming for parallel processing. IEEE Trans. C-27, 4, 289-296 (April 1978).

[Hearn 76] A. C. Hearn. Symbolic computation. Proc. CERN School of Computing, 201-211 (Sept. 1976).

[Henderson and Morris 76] P. Henderson and J. H. Morris, Jr. A lazy evaluator. Proc. 3rd ACM Conference on Principles of Programming Languages, 95-103 (Jan. 1976).

[Keller 75] R. M. Keller. Look-ahead processors. Computing Surveys, 7, 4, 177-195 (Dec. 1975).

[Keller 77] R. M. Keller. Semantics of parallel program graphs. University of Utah, Department of Computer Science, Tech. Rept. UUCS-77-110 (July 1977).

[Keller 78] R. M. Keller. An approach to determinacy proofs. University of Utah, Department of computer Science, Tech. Rept. UUCS-78-102 (March 1978)

[Lamport 74] L. Lamport. The parallel execution of DO loops. CACM, 17, 2, 83-93 (Feb. 1974).

[McCarthy 63] J. McCarthy. Towards a mathematical science of computation. Proc. IFIP '62, 21-28 (1963).

[Patil 67] S. Patil. An abstract parallel-processing system. M.S. Thesis. MIT, Department of Electrical Engineering (June 1967).

[Reddi and Feustel 78] S. S. Reddi and E. A. Feustel. A restructurable computer system. IEEE Trans. on computers, C-27, 1, 1-20 (Jan. 1978).

[Swan, *et al.* 77] R. J. Swan, S. H. Fuller, and D. P. Siewiorek. Cm* - A modular, multi-microprocessor. AFIPS Conference Proc., 46, 637-644 (June, 1977).