Massachusetts Institute of Technology
Artificial Intelligence Laboratory

AI Memo 796 September 1984

What a parallel programming language has to let you say

Alan Bawden
Philip E. Agre

Abstract: We have implemented in simulation a prototype language for the Connection Machine called CL1. CL1 is an extrapolation of serial machine programming language technology: in CL1 one programs the individual processors to perform local computations and talk to the communications network. We present details of the largest of our experiments with CL1, an interpreter for Scheme (a dialect of Lisp) that allows a large number of different Scheme programs to be run in parallel on the otherwise SIMD Connection Machine. Our aim was not to propose Scheme as a language for Connection Machine programming, but to gain experience using CL1 to implement an interesting and familar algorithm. Consideration of the difficulties we encountered led us to the conclusion that CL1 programs do not capture enough of the causal structure of the processes they describe. Starting from this observation, we have designed a successor language called CGL (for Connection Graph Language).

This report describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the laboratory's artificial intelligence research is provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-80-C-0505. Agre is supported by a fellowship from the Fannie and John Hertz Foundation.

Introduction

We have implemented a prototype language for the Connection Machine (henceforth *CM*) [11] called CL1 [4]. CL1 extrapolates serial machine programming language technology to the purposes of massively parallel machines. We observed that the individual CM processing elements could be modeled by a compiler in much the same way as a serial machine compiler models the target machine's CPU and registers. A CL1 program talks to the CM's communications network in much the same was as a conventional machine talks to its memory.

By experimenting with this language, we hoped to expose its inadequacies and learn from them. We present here the details of our largest experiment, an interpreter for Scheme [14], a dialect of Lisp. Our aim was not to propose Scheme as a language for CM programming, but to gain experience using CL1 to implement an interesting and familar algorithm.

Consideration of the difficulties we encountered has led us to the conclusion that CL1 programs do not capture enough of the causal structure of the processes they describe. Starting from this observation, we have designed a successor language called CGL (for Connection Graph Language), which we will describe in [5].

The Connection Machine

The CM is a massively parallel computer now being designed and built at the MIT Artificial Intelligence Laboratory. It consists of a large array of identical small processor *cells* connected by a communications network. The cells combine features of both the processors and memory words of traditional architectures. By distributing processing power across a large address space, the CM architecture avoids the bottleneck between processor and memory characteristic of conventional architectures [3].

Like the memory of a conventional machine, the CM is intended to have enough cells that the expense of allocating a new cell should be comparable to that of allocating a new word of memory on a conventional machine. The current design includes one million cells -- a 20-bit address space.

Like the CPU of a conventional machine, each cell has a few hundred bits of state. The cells do not have any local program storage or program counter. Instead, instructions are broadcast by the host machine to all cells at once over a common instruction bus. The CM is thus a SIMD (single-instruction, multiple-data) architecture, although the SIMD nature of the machine does not play any role in this paper (except in the section about efficiency considerations). Each cell has a serial ALU capable of executing any of a small number of operations. The programmer can specify that only a certain subset of the cells in the machine should execute a given instruction by making execution of that instruction conditional on the value of a status flag in each cell.

The cells are connected by a communications network. If cell A has the address of cell B in its local memory, then A can use the network to transmit a "message" of any length to B. Neither cell need know anything about the structure of the network in-between. This network's transmission protocol has been kept simple because of the sobering observation that one pays for any feature one adds to the cells a million times over. The hardware provides no mechanisms for message arbitration; there is no hardware support for queueing, for example. Because of this, CM programming languages must provide more useful message

sending abstractions built on the primitive foundation provided by the hardware.

Programming a Connection Machine

The CM was originally designed as a peripheral for storing and operating on semantic network databases. Since the CM's design was more-or-less finalized, we have been able to experiment with programming it (in microcode) for other applications. Here are the basics of CM programming as we now understand them; see [6] for more details.

In the CM, as in a conventional machine, one builds data structures by allocating cells and connecting them together by storing addresses of cells -- "pointers" -- in the memory of other cells. Since a pointer can *only* be used as an instruction to the communications network for delivering a message, the pattern of pointers in a CM might best be thought of as a virtual communications network. The ability to pass pointers from cell to cell as data can thus be thought of as the ability to dynamically reconfigure the network to suit the occasion. This ability to form new "connections" through the communications network, under software control, gives the Connection Machine its name.

Cells in the CM, like words of a conventional machine's memory, will typically be used for a number of different purposes in any particular computation. Some cells might be linked into a binary tree to represent a set. Others might act as record structures to represent objects with components. A space-ship might have a position, a velocity, and an owner.

The manipulations performed by a cell on the data in its local storage closely resemble those of a conventional general-register computer on the data kept in its register set. Calculations are performed on data, and the results are written back over data that is no longer needed. Temporary storage is allocated to contain intermediate results. Since there is no run-time storage management, the layout of local storage is static and must be established at compile-time. There is no natural way to implement stacks, arrays, or heaps. The programmer has a *complete model* of the usage of each bit in a cell's local storage.

Programmers most commonly deal with the lack of synchronization or queueing features in the message transmission facility by limiting the circulation of a cell's address. Adoption of appropriate conventions about the handling of pointers can guarantee that no more than one cell will ever wish to transmit messages to any given cell. If many cells might potentially wish to communicate with some target cell, then a tree of intermediate cells can be built to handle the "fan-in" of messages, effectively constructing a queue out of cells. (See [11].) Other higher-level protocols can be supported by appropriately coordinating message transmission, and by building auxiliary supporting structures out of cells. To a large extent, the design of algorithms for the CM is the design of such communications protocols.

Sometimes the representation of an object that the programmer wishes to treat atomically will require more storage than that of a single cell. In this case, several cells must be allocated and linked together, by a mutual exchange of addresses if nothing else. This forces the programmer to deal with *internal* communications within her data-structures, as well as external communications between them.

CL1 Summary

The chief motivation behind the design of CL1 was the observation that the storage management in a single CM processing element closely parallels management of the registers in a conventional general-register von Neumann machine. Many expressions in a conventional programming language can be compiled into "straight-line" code which uses only the fixed storage contained in the register file of a conventional machine. Since a single CM cell contains about the same amount of storage as a conventional register file, such code can be executed on a single CM cell. This suggested that we try programming individual cells in such a language. We can support message passing by simply adding primitive transmitting and receiving operators to our language.

Having chosen to view programming a CM in this manner, we built a compiler using off-the-shelf compiler technology. We drew on techniques from [12], as well as some more standard techniques such as those found in [2]. The resulting language, a dialect of Lisp resembling Scheme [14], is documented completely in [4].

CL1 programs are thought of as executed by *single* CM cells. This limits the kind of programs that can be executed. Since each cell has a fixed number of bits of state, programs requiring unbounded state will not be executable. CL1 disallows two features which in more powerful Lisps allow programs to employ unbounded state: recursion other than tail-recursion, and runtime closures. Truncated in this manner, CL1 cannot be used for programs that require more than a fixed amount of state, as determined at compile time. CL1 is basically a language for writing finite state machines.

In the CL1 model of a CM, every allocated cell is in some "state". Each state has associated with it a number of "state variables". The values of these variables at any cell in that state are stored in bits allocated from that cell's local storage. Also associated with each state is a straight-line (non-branching) piece of code to be run by any cell in that state. That code performs computations with the cell's state variables, computes a new state for it, and provides values for any new state variables.

It is the CL1 compiler's job to hide this state-machine model of program execution from the programmer wherever possible. The programmer normally expresses the behavior of cells in a flexible, Lisp-like language. However, sometimes the programmer must *explicitly* talk about a state with its associated variables and code. This is usually done using the DEFSTATE form, which allows a programmer to give a name to a state:

```
(DEFSTATE FACTORIAL-STATE (NUMBER)
  (LABELS ((FACT (LAMBDA (ACCUMULATOR COUNTER)
                   (IF (< COUNT 2) ACCUMULATOR
                       (FACT (* COUNTER ACCUMULATOR)
                             (1- COUNTER))))))
    (GO NEXT-STATE (FACT 1 NUMBER) NUMBER)))
```

This form defines a state named FACTORIAL-STATE, with a single state variable named NUMBER. If a cell is placed in state FACTORIAL-STATE and its state variable is initialized to contain some integer, that cell will proceed to compute the factorial of that number. When it is finished, it will set its state to NEXT-STATE, initializing its first state variable to contain the newly-computed factorial and its second state variable to contain the original number.

Notice that most of the body of a DEFSTATE form looks like ordinary Lisp code. This, after all, was the major goal in implementing CL1: to allow familiar tools to be applied to CM programming.

Support for message passing in CL1 is quite simple. A cell can have a number of *channels* to receive messages. The TRANSMIT function takes a pointer to a channel in another cell and a message and causes that message to arrive as input in that channel. The WITH-CHANNEL special form allocates a location within the executing cell to be a channel and creates a pointer to that location, e.g.,

```
(WITH-CHANNEL (CHANNEL POINTER)
  (MAKE-CELL NEW-CELL-STATE POINTER)
  (INPUT CHANNEL))
```

In this example, the variable POINTER will be bound to the new pointer and the variable CHANNEL will be bound to the "channel object" at which the pointer points. *A pointer points to a channel within a cell, not to the cell itself.* The location of the channel will be deallocated when the body of the WITH-CHANNEL form is finished executing. The cell will be able to read any message transmitted to it using this pointer by applying the INPUT function to the channel. In the example the executing cell creates a pointer-channel pair, allocates a new cell using the MAKE-CELL function (not a trivial operation; see [6]), places that new cell in the state NEW-CELL-STATE, initializes its single state variable to contain the new pointer, and then waits for a message to arrive from the new cell. A version of TRANSMIT, called TRANSMIT-WAIT, performs arbitration when more than one cell sends a message to the same channel at the same time. Each TRANSMIT-WAIT call hangs until the receiving cell's INPUT call gets around to that message. The TRANSMIT-WAIT construct is not a hardware primitive; it compiles to a complex protocol.

Since CL1 has no theory of data other than providing a pointer datatype to represent CM addresses, the values manipulated by CL1 programs in the present implementation are simply MacLisp objects. Data types other than CL1 pointers are left to the programmer to simulate as best she can. Short fixed-length lists, for example, are commonly used to implement records. The programmer is on her honor not to use these objects in ways that are clearly impossible on an actual CM.

Coding cliches

In the course of writing the first few CL1 programs, a number of conventions and coding cliches arose that we have tried to capture with macros. Here we present the ones that are convenient to explain outside the context of the Scheme interpreter described in the next section.

Every channel in a cell uses one of two message-passing contracts. A *continuation* is a channel, typically created as a place for another cell to send the result of its calculation, to which there is exactly one pointer in the system. One uses TRANSMIT to send a message to a continuation. A *command channel* is used by a cell to offer some contract to any taker. Messages to command channels, called *commands*, are sent using TRANSMIT-WAIT. A command has a *command type* and a fixed number of *fields*, the first of which must be a continuation to which the response should be sent.

When a cell creates a new cell with MAKE-CELL, the following scheme is often used to establish communication between the new cell and its owner. The new cell will have among its initial state variables one called OWNER, which is initialized to a pointer to an input channel in the cell that made it. The newly

created cell is expected to send to the owner's channel a pointer to a channel of its own.

Many types of cells are quite passive, merely handling one command after another. Such *command-loop cells* recall the "objects" of languages like Smalltalk. As an example, here is a piece of CL1 code taken from the Scheme interpreter that illustrates these three conventions:

```
(DEFSTATE CONS-NODE (OWNER CAR-PART CDR-PART)
  (WITH-CHANNEL (COMMAND-CHANNEL COMMAND-POINTER)
    (TRANSMIT OWNER COMMAND-POINTER)
    (DO ((COMMAND (INPUT COMMAND-CHANNEL)
                  (INPUT COMMAND-CHANNEL)))
        (NIL)
      (DISPATCH COMMAND
        ((GET-CAR CONTINUATION)
         (TRANSMIT CONTINUATION CAR-PART))
        ((GET-CDR CONTINUATION)
         (TRANSMIT CONTINUATION CDR-PART))
        ))))
```

A CONS-NODE sends its owner a pointer to its command channel and then enters an eternal loop, dispatching off the type of each new command and transmitting the appropriate answer to the command's continuation. The DEFOBJECT macro can be used to suppress the first four lines of this definition:

```
(DEFOBJECT CONS-NODE (CAR-PART CDR-PART) (COMMAND)
  (DISPATCH COMMAND
    ((GET-CAR CONTINUATION) (TRANSMIT CONTINUATION CAR-PART))
    ((GET-CDR CONTINUATION) (TRANSMIT CONTINUATION CDR-PART))
    ))
```

The DISPATCH form dispatches on the type of a command, locally binding the fields of the command to variables. The clause of a DISPATCH corresponding to a given type of command is commonly called that command's *method.* The code for CONS-NODE has a GET-CAR method and a GET-CDR method.

One commonly sends a cell a command and calls INPUT on the provided continuation to await a response. The DEFCOMMAND macro can be used to define a functional form to capture this cliche. For example, (DEFCOMMAND GET-CAR (CONS-NODE)) allows one to obtain the CAR-PART of a CONS-NODE by saying (GET-CAR *cell*). This form will expand into:

```
(WITH-CHANNEL (TEMPORARY-CHANNEL TEMPORARY-POINTER)
  (TRANSMIT-WAIT cell
    (MAKE-COMMAND 'GET-CAR TEMPORARY-POINTER))
  (INPUT TEMPORARY-CHANNEL))
```

Notice that a cell that uses GET-CAR or any other DEFCOMMAND form must sit waiting in INPUT until the target cell computes and returns the answer.

The DEFMAKER macro defines a functional form to create a cell that employs the OWNER convention, e.g., (DEFMAKER CONS-NODE (CAR-PART CDR-PART)) lets one obtain a pointer to the command channel of a newly created CONS-NODE by saying (MAKE-CONS-NODE *car-part cdr-part*). This form will expand into:

```
(WITH-CHANNEL (TEMPORARY-CHANNEL TEMPORARY-POINTER)
  (MAKE-CELL CONS-NODE TEMPORARY-POINTER car-part cdr-part)
  (INPUT TEMPORARY-CHANNEL))
```

Scheme interpreter

To see what programming in CL1 is like, we wrote an interpreter for Scheme, a lexically scoped and tail-recursive dialect of Lisp. The most obvious interest of this exercise is the promise of being able to employ arbitrarily differentiated processing on the putatively SIMD Connection Machine: by multiplexing the CM's instruction stream among the different kinds of objects in a running Scheme program, thousands of different Scheme programs can be executed in parallel at the cost of a moderate constant factor slow-down. (A similar tack is taken in [9].) Even so, we don't consider Scheme to be particularly well-suited for programming parallel machines: it provides no better account than the average language of the important linguistic issues of how best to express methods of process decomposition and synchronization.

We chose Scheme interpretation as our first experiment in CL1 programming not because Scheme is the ultimate parallel programming language but because the Scheme interpreter is an interesting and relatively simple algorithm that we understand thoroughly. There is an extensive literature and culture of the implementation of Scheme and its neighbors in language-space (see [15]). The Turing-universality of the algorithm provides some vague promise that a wide variety of programming issues must be addressed in its implementation. A less familiar programming problem would have introduced uncertainties unrelated to the linguistic issues that were our real interest.

We assume that the reader has a reasonable grounding in Scheme and the issues involved in its interpretation; see [14] or [1]. For our purposes, the most important features of a Scheme interpreter are:

- The language is lexically scoped (as is, for example, Algol-60). A procedure object is created by combining a pointer to the body of the procedure with a pointer to the current environment. When a procedure object is applied to arguments, the body is interpreted in an environment constructed by extending the procedure object's environment with the new bindings of the procedure's formal parameters.

- Procedure objects are first-class data; they can be passed as arguments, returned as values, and incorporated into compound data structures. That one should be able to make general use of procedure objects without excessive penalty is part of the Scheme programming philosophy and a strict constraint on the Scheme implementor.

- A Scheme interpreter must be *tail-recursive*. While a recursive procedure must wait for the result from its recursive call so it can perform additional operations with that result, a tail-recursive procedure, having no additional operations to perform, need not push a return address. (See [13].) The interpreter, when calling itself on a subexpression of a given expression, saves only the information that will be needed after the evaluation of the subexpression has been completed. In addition to allowing users to use procedure calls liberally, this allows efficient implementation of all common control constructs in terms of conditional evaluation and the procedure call.

In our CL1 Scheme interpreter, a program is represented as a network of cells that is produced as a parse tree by a syntaxer (roughly speaking, a parser). For instance, Figure 1 portrays the network corresponding to the expression:

```
(DEFINE MAKE-INCREMENTER
  (LAMBDA (INCREMENT)
    (LAMBDA (X)
      (PLUS INCREMENT X))))
```

To evaluate a Scheme expression in an environment, one sends it an EVALUATE command, providing a continuation and the environment, implemented as a chain of FRAME and ALIST cells. Figure 2 shows a portion of the global environment after the definition of MAKE-INCREMENTER has been evaluated by sending such a command to the DEFINITION cell at the top of the tree. Figure 3 shows the environment after the form (DEFINE INCREMENTER (MAKE-INCREMENTER 7)) has been syntaxed and evaluated.

The contract of an EVALUATE command is that the result of evaluting the receiving expression in the indicated environment should eventually be sent to the indicated continuation. On a serial machine, the Scheme evaluator dispatches on the type of the cell being evaluated, but in our interpreter each object knows how to evaluate itself. (This is reminiscent of what on serial machines is called object-oriented programming.) There are eight types of cells in a program:

CONDITIONALs, as in (IF *predicate then else*)
VARIABLEs, such as CAR and PATTERN
DEFINITIONs, as in (DEFINE *symbol expression*)
LAMBDA-EXPRESSIONs, as in (LAMBDA *variables body*)
PROCEDURE-APPLICATIONs and OPERAND-CONSes, as in (*procedure . operands*)
SEQUENCEs, as in (SEQUENCE . *expressions*)
PARALLELs, as in (PARALLEL . *expressions*)
CONSTANTs, such as (QUOTE (EATS KITTY FISH)) and 23

As an example, consider the CONDITIONAL cell type. It has state variables called PREDICATE, THEN, and ELSE, each pointing at the cell for some other Scheme expression. Here is one way that the CONDITIONAL cell could have been implemented:

```
(DEFOBJECT CONDITIONAL (PREDICATE THEN ELSE) (COMMAND)
  (DISPATCH COMMAND
    ((EVALUATE CONTINUATION ENVIRONMENT)
     (TRANSMIT-WAIT (IF (EVALUATE PREDICATE ENVIRONMENT)
                        THEN
                        ELSE)
       COMMAND))
    ))
```

A CONDITIONAL cell accepts a stream of EVALUATE commands. It uses the EVALUATE form defined using DEFCOMMAND to find the value of the predicate and hands the result to CL1's IF, which determines whether THEN or ELSE will be evaluated to provide the result. The winning expression is sent the command the CONDITIONAL received, and so will pass its result on to that command's continuation itself. Meanwhile, the CONDITIONAL cell can move on to new EVALUATE commands. This continuation-passing trick (cf. [10])

Figure 1. CM representation of a Scheme program.

The syntaxer for our CL1 Scheme interpreter lays out a Scheme program in the CM as a tree of cells, one for each part of the code. To evaluate a piece of Scheme code in some environment, one sends the cell at the top of the tree a command of the form (EVALUATE *continuation environment*). The network corporately guarantees that the right answer will eventually arrive at the continuation.

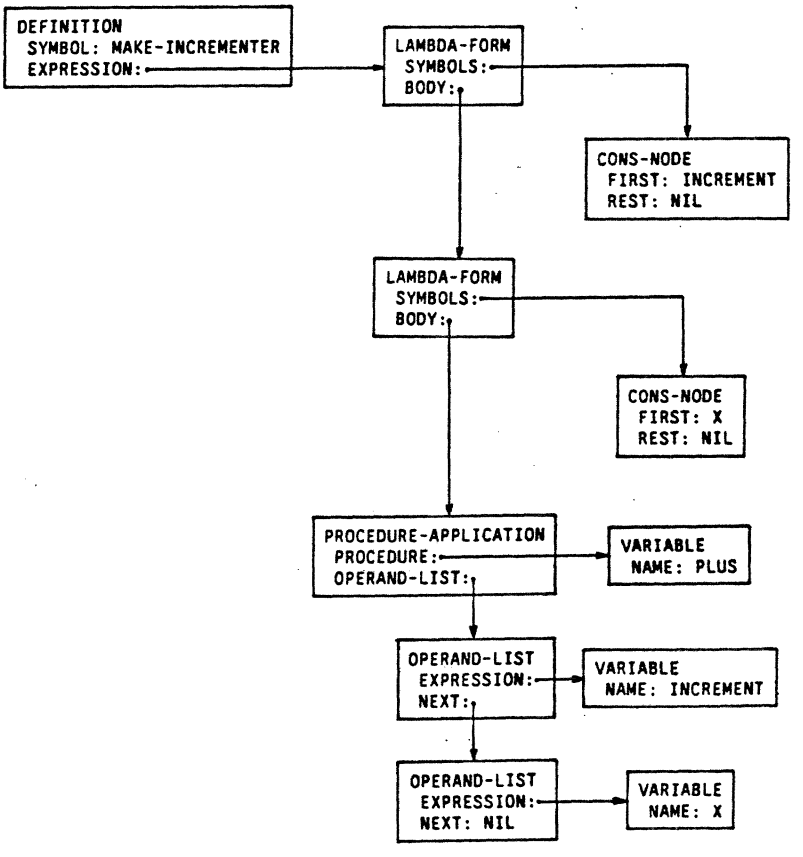


Figure 2. After defining MAKE-INCREMENTER.

A CL1 Scheme environment is organized as a sequence of FRAMEs, each of which is composed of a sequence of binding pairs (a chain of ALIST cells). LOOKUP and SET-VARIABLE commands propagate from a FRAME to its chain of ALIST cells, and from the last ALIST to the following FRAME. Procedures are data objects like any others and a procedure is normally fetched by looking its name up in the current environment. The object that embodies the procedure is a cell of type CLOSURE. A CLOSURE is created by the evaluation of a LAMBDA-EXPRESSION and contains not only the parameter list and body of the LAMBDA-EXPRESSION but also the environment that was current when the LAMBDA-EXPRESSION was evaluated. The body will be evaluated in this environment when the CLOSURE is applied to arguments.

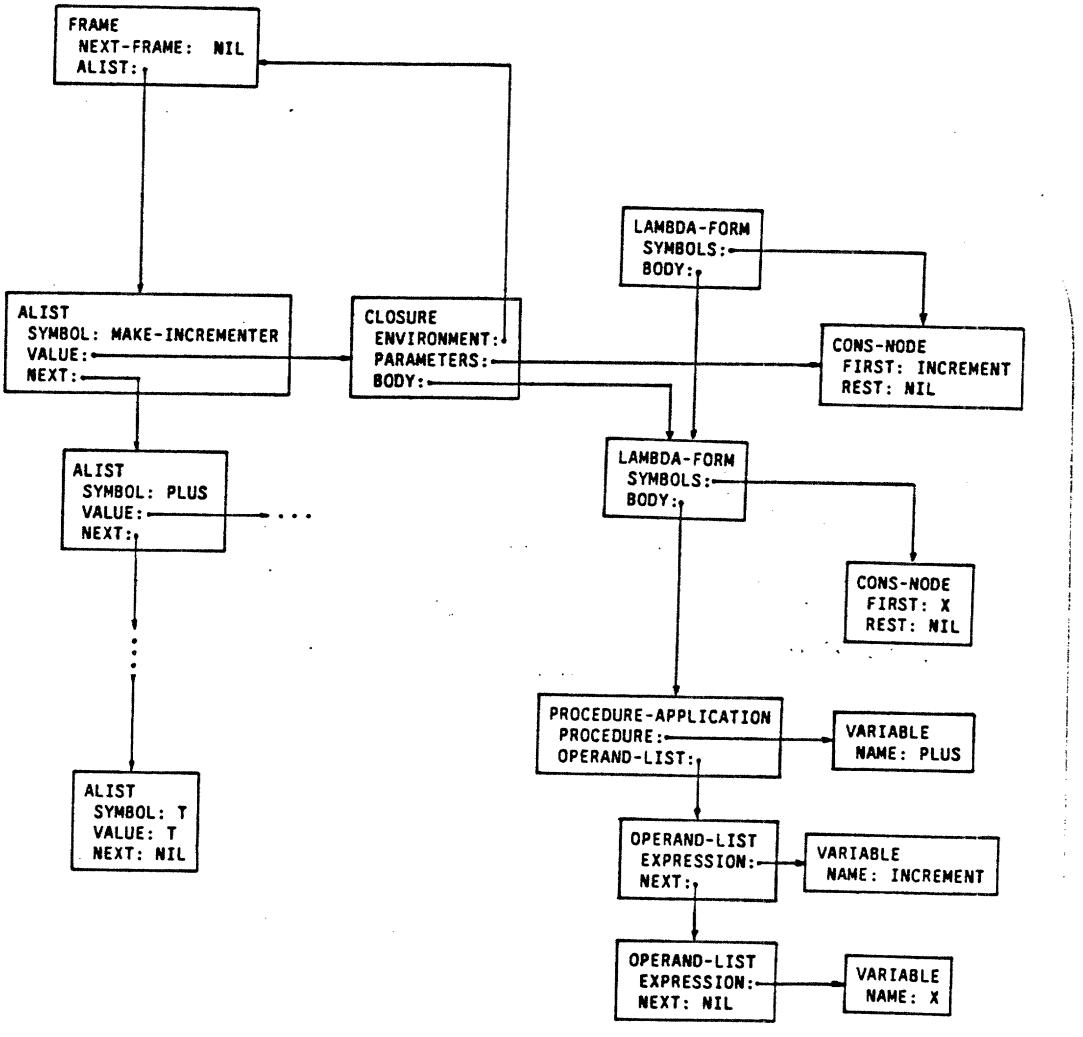
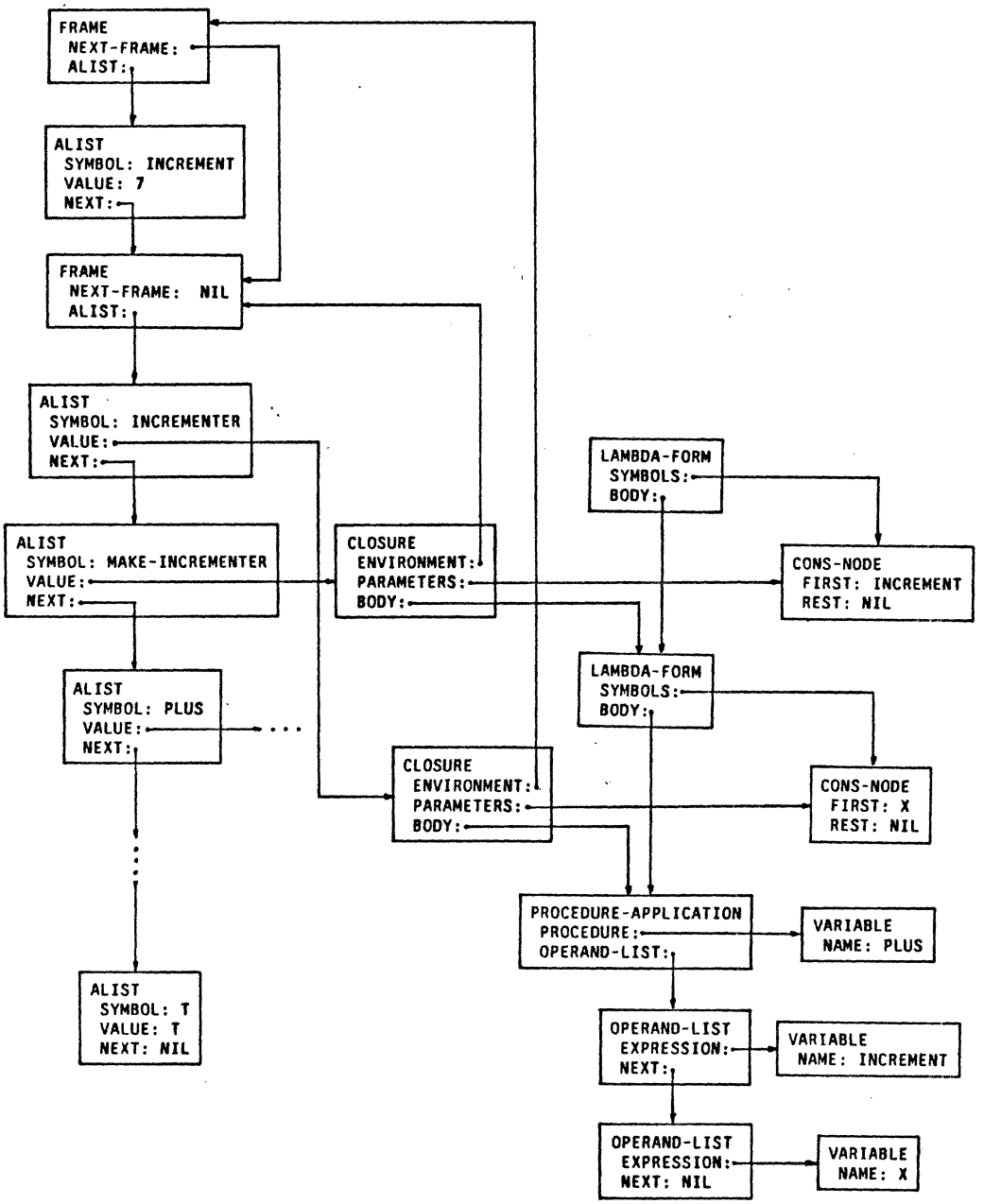


Figure 3. After defining INCREMENTER.

When (DEFINE INCREMENTER (MAKE-INCREMENTER 7)) is evaluated, a new ALIST cell is created in the global environment; this cell binds INCREMENTER to a new CLOSURE. A new FRAME is created to serve as the environment of that closure, in which X has the value 7. The new FRAME's parent environment is the global environment.



corresponds to the tail-recursive nature of the evaluation of conditionals in serial-machine Scheme interpreters.

Yet conditional evaluation isn't quite tail-recursive, since the CONDITIONAL cell must wait on the result of the evaluation of the predicate. If that predicate is a complicated expression, a large number of new commands could back up in the CONDITIONAL's input queue. Not only that, but if the evaluation of the predicate involves evaluating that same CONDITIONAL in some environment, the system will deadlock with the CONDITIONAL and its predicate waiting on one another.

Therefore, the CONDITIONAL creates a temporary cell to wait on the value of the predicate and pass the EVALUATE command on to THEN or ELSE:

```
(DEFOBJECT CONDITIONAL (PREDICATE THEN ELSE) (COMMAND)
  (DISPATCH COMMAND
    ((EVALUATE CONTINUATION ENVIRONMENT)
     (TRANSMIT-WAIT PREDICATE
       (MAKE-COMMAND 'EVALUATE
         (FORK (RESULT-CHANNEL)
           (TRANSMIT-WAIT (IF (INPUT RESULT-CHANNEL) THEN ELSE)
             COMMAND))
         ENVIRONMENT)))
    ))
```

(MAKE-COMMAND makes commands; it shouldn't be confused with the forms created by DEFMAKER.) The FORK macro suppresses the details of making a temporary cell to wait on the predicate outcome and then perform the inner TRANSMIT-WAIT. The user thinks of the FORK form as returning a pointer to the temporary cell's RESULT-CHANNEL; this is the continuation for the evaluation of the predicate. Here is the code into which the FORK form expands:

```
(WITH-CHANNEL (TEMP-CHANNEL TEMP-POINTER)
  (MAKE-CELL (KAPPA (TEMP-OWNER)
               (WITH-CHANNEL (RESULT-CHANNEL RESULT-POINTER)
                 (TRANSMIT TEMP-OWNER RESULT-POINTER)
                 (TRANSMIT-WAIT (IF (INPUT RESULT-CHANNEL)
                                    THEN
                                    ELSE)
                   COMMAND)))
             TEMP-POINTER)
  (INPUT TEMP-CHANNEL))
```

CL1's KAPPA allows one to create anonymous states by analogy to the Scheme LAMBDA construct for creating anonymous procedures. The temporary cell, then, is started up in a state in which it creates a channel, sends a pointer to it to its owner in the traditional manner, and waits for the outcome of the predicate to arrive in that channel. After sending the original COMMAND to either THEN or ELSE the temporary cell becomes garbage. (CL1 extends the analogy between KAPPA and LAMBDA by making sure that variables free in a KAPPA form, in this case COMMAND, THEN, and ELSE, are automatically transmitted to any cell created in the state defined by the form.)

The temporary cell created in this process is analogous to the stack space that a serial-machine Scheme

interpreter must occupy during its recursive call on the predicate of a conditional. Such cells are used in several places in the CL1 Scheme interpreter, each of them corresponding to a non-tail-recursive call in the usual meta-circular (that is, written in Scheme) Scheme interpreter (see [15]). Tail-recursion in the CL1 Scheme interpreter both increases the throughput of cells by not making them wait for answers and saves space by not making them create temporary cells to do that waiting.

Thus the evaluation of a CONDITIONAL cell involves little more than passing the EVALUATE command off to other cells. A similar tack is taken by PROCEDURE-APPLICATION cells:

```
(DEFOBJECT PROCEDURE-APPLICATION (PROCEDURE OPERAND-LIST) (COMMAND)
  (DISPATCH COMMAND
    ((EVALUATE CONTINUATION ENVIRONMENT)
     (TRANSMIT-WAIT PROCEDURE
       (MAKE-COMMAND 'EVALUATE
         (FORK (RESULT-CHANNEL)
           (TRANSMIT-WAIT (INPUT RESULT-CHANNEL)
             (MAKE-COMMAND 'EVLIS
               CONTINUATION
               OPERAND-LIST
               ENVIRONMENT)))
         ENVIRONMENT)))
    ))
```

When a PROCEDURE-APPLICATION cell receives an EVALUATE command, it must evaluate the PROCEDURE expression in the provided environment and then tell the resulting closure to evaluate the operands and apply itself to the resulting arguments. Rather than wait for that evaluation of PROCEDURE to return a value, though, it uses the FORK form to create a temporary cell to do the waiting and send the EVLIS command to the closure. Once the forked cell finishes sending the EVLIS command, it has no more code to execute and so passes into the null state and is deallocated.

A closure in Scheme is an object created by the evaluation of a LAMBDA expression. It contains the parameters and body given in the LAMBDA together with the environment that was current when the LAMBDA expression was evaluated. When a procedure is applied to operands, the operands are evaluated in the current environment; once the parameters have been bound to the resulting arguments, the body of the procedure is evaluated in the environment that is stored in the closure. That the closure and not the caller determines the environment of evaluation for the body of a newly called procedure is the essence of Scheme's lexical scope, as opposed to the dynamic scope of traditional Lisp.

In our implementation, a cell of type CLOSURE is created by the evaluation of a LAMBDA-EXPRESSION object in some environment:

```
(DEFOBJECT LAMBDA-EXPRESSION (SYMBOLS BODY) (COMMAND)
  (DISPATCH COMMAND
    ((EVALUATE CONTINUATION ENVIRONMENT)
     (MAKE-CELL CLOSURE CONTINUATION ENVIRONMENT SYMBOLS BODY))
    ))
```

A CLOSURE processes EVLIS commands sent to it by PROCEDURE-APPLICATIONs. The fields of an EVLIS command are: (1) the continuation to which the eventual result is to be sent, (2) the unevaluated

operand list, and (3) the environment in which these operands are to be evaluated:

```
(DEFOBJECT CLOSURE (ENVIRONMENT PARAMETERS BODY) (COMMAND)
  (DISPATCH COMMAND
    ((EVLIS CONTINUATION OPERANDS OPERAND-ENVIRONMENT)
     (TRANSMIT-WAIT BODY
       (MAKE-COMMAND 'EVALUATE
         CONTINUATION
         (MAKE-FRAME ENVIRONMENT
                     PARAMETERS
                     OPERANDS
                     OPERAND-ENVIRONMENT))))
    ))
```

Upon receiving and destructuring the EVLIS command, the CLOSURE object extends the environment and tail-recursively initiates the evaluation of the body in the resulting environment. The MAKE-FRAME form, which initiates the process of evaluating the operands and extending the environment, does not wait for that process to finish; instead it immediately returns a pointer to the FRAME that will lie at the head of the extended environment.

It is the responsibility of a FRAME, then, to see that the operands are evaluated and bound to symbols in fresh ALIST cells. After identifying itself to its owner, it starts a BIND command propagating down the operand list. This results in a pointer to the first of a newly created ALIST chain being returned to it, whereupon it settles into a command loop.

```
(DEFSTATE FRAME (OWNER NEXT-FRAME PARAMETERS OPERANDS ENVIRONMENT)
  (WITH-CHANNEL (COMMAND-CHANNEL COMMAND-POINTER)
    (TRANSMIT OWNER COMMAND-POINTER)
    (LET ((INITIAL-ALIST
            (IF OPERANDS
                (BIND OPERANDS PARAMETERS ENVIRONMENT NEXT-FRAME)
                NEXT-FRAME)))
      (ITERATE FRAME-LOOP ((ALIST INITIAL-ALIST))
        (LET ((COMMAND (INPUT COMMAND-CHANNEL)))
          (DISPATCH COMMAND
            ((LOOKUP CONTINUATION SYMBOL)
             (TRANSMIT-WAIT ALIST COMMAND)
             (FRAME-LOOP ALIST))
            ((SET-VARIABLE CONTINUATION SYMBOL VALUE)
             (TRANSMIT-WAIT ALIST COMMAND)
             (FRAME-LOOP ALIST))
            ((DEFINE CONTINUATION SYMBOL VALUE)
             (TRANSMIT CONTINUATION T)
             (FRAME-LOOP (MAKE-ALIST SYMBOL VALUE ALIST)))
            ))))))
```

The evaluation of operand lists is a little tricky and not entirely pleasant. An operand list is implemented as a chain of OPERAND-CONS cells, each representing one of the operands. When an OPERAND-CONS cell receives a BIND message, it evaluates its operand expressions in the provided environment and builds ALIST cells for the results and the formal parameters.

The code for OPERAND-CONSes and ALISTs requires a fair bit of explanation. The EXPRESSION variable of an OPERAND-CONS points to (the command channel of) the operand, which is some arbitrary expression. The NEXT variable contains either NIL, indicating that this is the last operand, or a pointer to another OPERAND-CONS cell. When an OPERAND-CONS receives a BIND command, it propagates it down the chain, making a new ALIST cell at every step:

```
(DEFOBJECT OPERAND-CONS (EXPRESSION NEXT) (COMMAND)
  (DISPATCH COMMAND
    ((BIND CONTINUATION PARAMETERS ENVIRONMENT NEXT-FRAME)
     (IF (NULL NEXT)
         (MAKE-CELL LAST-PROTO-ALIST
                    CONTINUATION
                    (GET-CAR PARAMETERS)
                    EXPRESSION
                    ENVIRONMENT
                    NEXT-FRAME)
         (TRANSMIT-WAIT NEXT
           (MAKE-COMMAND 'BIND
             (MAKE-PROTO-ALIST CONTINUATION
                               (GET-CAR PARAMETERS)
                               EXPRESSION
                               ENVIRONMENT)
             (GET-CDR PARAMETERS)
             ENVIRONMENT
             NEXT-FRAME))))
    ))
```

Before understanding how OPERAND-CONSes work, consider one way they *might have* worked. An OPERAND-CONS cell could:

(1) evaluate its operand,
(2) wait while subsequent operands are evaluated to yield the tail of the new chain of binding pairs,
(3) attach a new ALIST cell that binds its symbol to its value to the front of that tail, and
(4) return the newly created ALIST to the continuation, be it the previous OPERAND-CONS cell or the CLOSURE fork that is waiting on the new environment.

This might work, but it would require the OPERAND-CONS cells at the front to spend almost all their time waiting when they could be evaluating their expressions in the service of other CLOSUREs. Other kinds of cells, like CLOSUREs, solved this problem by using the FORK form to make a temporary cell to do the waiting. But in the case of OPERAND-CONSes there is no need to go to the extra expense: the newly created ALIST cell can do the waiting itself. This trick is the reason why it is OPERAND-CONS cells that evaluate arguments rather than PROCEDURE-APPLICATION cells, as a direct translation of the usual meta-circular Scheme interpreter into CL1 would have it.

There are two cases. A new ALIST can wait on the results of operand evaluation or on both that evaluation and the construction of the remaining ALISTs, depending on whether it is the last ALIST in that frame. Consequently, there are two different states in which an ALIST cell can be created, PROTO-ALIST and LAST-PROTO-ALIST:

```
(DEFSTATE LAST-PROTO-ALIST (OWNER SYMBOL EXPRESSION ENVIRONMENT NEXT)
  (GO ALIST OWNER SYMBOL (EVALUATE EXPRESSION ENVIRONMENT) NEXT))

(DEFSTATE PROTO-ALIST (NEXT-SOURCE OWNER SYMBOL EXPRESSION ENVIRONMENT)
  (WITH-CHANNEL (EVAL-CHANNEL EVAL-POINTER)
    (TRANSMIT-WAIT EXPRESSION
      (MAKE-COMMAND 'EVALUATE EVAL-POINTER ENVIRONMENT))
    (WITH-CHANNEL (NEXT-CHANNEL NEXT-POINTER)
      (TRANSMIT NEXT-SOURCE NEXT-POINTER)
      (GO ALIST
          OWNER
          SYMBOL
          (INPUT EVAL-CHANNEL)
          (INPUT NEXT-CHANNEL)))))

(DEFSTATE ALIST (OWNER BOUND-SYMBOL INITIAL-VALUE NEXT)
  (WITH-CHANNEL (COMMAND-CHANNEL COMMAND-POINTER)
    (TRANSMIT OWNER COMMAND-POINTER)
    (ITERATE ALIST-LOOP ((VALUE INITIAL-VALUE))
      (LET ((COMMAND (INPUT COMMAND-CHANNEL)))
        (DISPATCH COMMAND
          ((LOOKUP CONTINUATION SYMBOL)
           (COND ((EQ SYMBOL BOUND-SYMBOL)
                  (TRANSMIT CONTINUATION VALUE))
                 ((NOT (NULL NEXT))
                  (TRANSMIT-WAIT NEXT COMMAND))
                 (T
                  (FORMAT T
                    "UNBOUND SCHEME VARIABLE: ~S"
                    SYMBOL)))
           (ALIST-LOOP VALUE))
          ((SET-VARIABLE CONTINUATION SYMBOL NEW-VALUE)
           (COND ((EQ SYMBOL BOUND-SYMBOL)
                  (TRANSMIT CONTINUATION T)
                  (ALIST-LOOP NEW-VALUE))
                 ((NOT (NULL NEXT))
                  (TRANSMIT-WAIT NEXT COMMAND)
                  (ALIST-LOOP VALUE))
                 (T
                  (FORMAT T
                    "ATTEMPT TO SET AN UNBOUND VARIABLE: ~S"
                    SYMBOL)
                  (ALIST-LOOP VALUE))))
          )))))
```

There is quite a bit going on here. Here is the image: as the BIND command propagates down the chain of OPERAND-CONS cells, a parallel chain of PROTO-ALIST cells is created, each of which has initiated the evaluation of its corresponding operand and is waiting for the next ALIST cell along to identify itself. When the BIND command reaches the last OPERAND-CONS cell along, control "turns around" at a LAST-PROTO-ALIST cell, propagating back along the chain of proto-alist cells, each of which initializes itself in turn by moving into the ALIST state. (Operands are thus evaluated in parallel.) The process is completed when the first ALIST cell in the chain identifies itself to the new FRAME.

We used such a peculiar pattern of message sending and receiving to implement operand list evaluation so that no cell that could be doing useful work has to wait for anything. In particular, we don't want the OPERAND-CONS that creates a new binding pair to have to wait for all the downstream binding pairs to be created first. But each ALIST cell requires a pointer to its successor. Consequently, when an OPERAND-CONS cell creates a PROTO-ALIST cell, the PROTO-ALIST cell returns to it a pointer (called NEXT-POINTER) to a channel (called NEXT-CHANNEL) to which (a pointer to the command channel of) the next ALIST cell along should be sent once it is known. This pointer is sent along to the next OPERAND-CONS as part of the BIND message. The boundary case of this process is at the last OPERAND-CONS, for which the correct value of the new ALIST cell's NEXT variable is known. (This value is not NIL, but rather the next frame in the environment, where failed LOOKUP and SET-VARIABLE commands should be propagated.) Therefore, LAST-PROTO-ALIST cells perform only half of this complex protocol: they wait on the evaluation of the operand, but are given the NEXT right away rather than having to wait for it as well.

Once an ALIST cell has entered its command loop, it can accept two types of commands -- SET-VARIABLE and LOOKUP, with the obvious semantics. The LOOKUP command is used by VARIABLE cells:

```
(DEFOBJECT VARIABLE (NAME) (COMMAND)
  (DISPATCH COMMAND
    ((EVALUATE CONTINUATION ENVIRONMENT)
     (TRANSMIT-WAIT ENVIRONMENT
       (MAKE-COMMAND 'LOOKUP CONTINUATION NAME)))
    ))
```

The only obscure point in the ALIST command loop is the SET-VARIABLE command's returning a value. This satisfies the contract of a command and also informs any caller who might be curious that the setting has been completed and whether it succeeded. One cannot set or look up a variable in an environment in which it is not bound. The calls to FORMAT would not carry over to a CM implementation, which would have to have its own ways of signalling errors. Support for error-handling in CM languages is as yet poorly understood. This concludes our explanation of procedure application in the CL1 Scheme interpreter.

SEQUENCE and PARALLEL constructs differ only in their attitude toward the result of executing the first of the two expressions, LHS and RHS. (If there are more than two forms in a SEQUENCE or PARALLEL form, the syntaxer will make a chain of SEQUENCE or PARALLEL cells.) A SEQUENCE must set up a FORK to await the completion of the evaluation of the LHS before sending the EVALUATE command on to the RHS. A PARALLEL, on the other hand, tells the LHS to send its result to a bit bucket and then immediately passes the EVALUATE command to the RHS. (CL1 does not support bit buckets specially, so there is a kind of cell called BIT-BUCKET just for this purpose.)

```
(DEFOBJECT SEQUENCE (LHS RHS) (COMMAND)
  (DISPATCH COMMAND
    ((EVALUATE CONTINUATION ENVIRONMENT)
     (TRANSMIT-WAIT LHS
       (MAKE-COMMAND 'EVALUATE
         (FORK (RESULT-CHANNEL)
           (INPUT RESULT-CHANNEL)
           (TRANSMIT-WAIT RHS COMMAND))
         ENVIRONMENT)))
    ))

(DEFOBJECT PARALLEL (LHS RHS) (COMMAND)
  (DISPATCH COMMAND
    ((EVALUATE CONTINUATION ENVIRONMENT)
     (TRANSMIT-WAIT LHS
       (MAKE-COMMAND 'EVALUATE (MAKE-BIT-BUCKET) ENVIRONMENT))
     (TRANSMIT-WAIT RHS COMMAND))
    ))

(DEFOBJECT BIT-BUCKET () (COMMAND))
```

Constants are the simplest expressions to evaluate. Upon receiving an EVALUATE message, a CONSTANT cell simply sends its constant back to the continuation:

```
(DEFOBJECT CONSTANT (DATA) (COMMAND)
  (DISPATCH COMMAND
    ((EVALUATE CONTINUATION ENVIRONMENT)
     (TRANSMIT CONTINUATION DATA))
    ))
```

All that remains is the initialization of the run-time environment. There is a separate cell type for each primitive the language supports, and these are bound to the appropriate symbols in the initial environment. The initial environment is set to the MacLisp variable GLOBAL-ENVIRONMENT using a sequence of calls to the DEFINE form (which was created by (DEFCOMMAND DEFINE (FRAME SYMBOL EXPRESSION))):

```
(DEFSTATE INITIALIZE-GLOBAL-ENVIRONMENT ()
  (LET ((GLOBAL (MAKE-FRAME NIL NIL NIL NIL)))
    (SET 'GLOBAL-ENVIRONMENT GLOBAL)
    (DEFINE GLOBAL 'CONS (MAKE-PRIMITIVE-CONS-CLOSURE))
    (DEFINE GLOBAL 'CAR (MAKE-PRIMITIVE-CAR-CLOSURE))
    (DEFINE GLOBAL 'CDR (MAKE-PRIMITIVE-CDR-CLOSURE))
    (DEFINE GLOBAL 'EQ? (MAKE-PRIMITIVE-EQ?-CLOSURE))
    (DEFINE GLOBAL '1+ (MAKE-PRIMITIVE-INCREMENT-CLOSURE))
    (DEFINE GLOBAL '1- (MAKE-PRIMITIVE-DECREMENT-CLOSURE))
    (DEFINE GLOBAL 'ZERO? (MAKE-PRIMITIVE-ZEROP-CLOSURE))
    (DEFINE GLOBAL '+ (MAKE-PRIMITIVE-PLUS-CLOSURE))
    (DEFINE GLOBAL '* (MAKE-PRIMITIVE-TIMES-CLOSURE))
    (DEFINE GLOBAL 'LIST? (MAKE-PRIMITIVE-LISTP-CLOSURE))
    (DEFINE GLOBAL 'SYMBOL? (MAKE-PRIMITIVE-SYMBOLP-CLOSURE))
    (DEFINE GLOBAL 'PRINC (MAKE-PRIMITIVE-PRINC-CLOSURE))
    (DEFINE GLOBAL 'NIL 'NIL)
    (DEFINE GLOBAL 'T 'T)))
```

The definitions of PRIMITIVE-CAR-CLOSURE and PRIMITIVE-CDR-CLOSURE illustrate two different ways of writing unary primitives. Each is defined using DEFOBJECT and uses an ordinary command loop, and each forks off a temporary cell to wait on operand evaluation. The difference is that while the fork created by PRIMITIVE-CAR-CLOSURE waits for the result of an EVALUATE-FIRST request sent to the operand list by the closure, the fork created by PRIMITIVE-CDR-CLOSURE does all of the work itself:

```
(DEFOBJECT PRIMITIVE-CAR-CLOSURE () (COMMAND)
  (DISPATCH COMMAND
    ((EVLIS CONTINUATION OPERAND-LIST OLD-ENVIRONMENT)
     (TRANSMIT-WAIT OPERAND-LIST
       (MAKE-COMMAND 'EVALUATE-FIRST
         (FORK (RESULT-CHANNEL)
           (TRANSMIT-WAIT (INPUT RESULT-CHANNEL)
             (MAKE-COMMAND 'GET-CAR CONTINUATION)))
         OLD-ENVIRONMENT)))
    ))

(DEFOBJECT PRIMITIVE-CDR-CLOSURE () (COMMAND)
  (DISPATCH COMMAND
    ((EVLIS CONTINUATION OPERAND-LIST OLD-ENVIRONMENT)
     (MAKE-CELL (KAPPA ()
                  (TRANSMIT-WAIT (EVALUATE-FIRST OPERAND-LIST
                                                 OLD-ENVIRONMENT)
                    (MAKE-COMMAND 'GET-CDR CONTINUATION)))))
    ))
```

The asymmetry is for only expository purposes; each definition could be written either way. The other primitive closures are turned out by macros operating on these models.

Primitive procedures do not construct alists, but rather send EVALUATE-FIRST and EVALUATE-SECOND commands to the operand lists they are given. The DISPATCH clauses for these commands were omitted in the definition of OPERAND-CONS given above.

```
(DEFOBJECT OPERAND-CONS (EXPRESSION NEXT) (COMMAND)
  (DISPATCH COMMAND
    ((BIND CONTINUATION PARAMETERS ENVIRONMENT NEXT-FRAME)
     ...)
    ((EVALUATE-FIRST CONTINUATION ENVIRONMENT)
     (TRANSMIT-WAIT EXPRESSION
       (MAKE-COMMAND 'EVALUATE CONTINUATION ENVIRONMENT)))
    ((EVALUATE-SECOND CONTINUATION ENVIRONMENT)
     (TRANSMIT-WAIT NEXT
       (MAKE-COMMAND 'EVALUATE-FIRST CONTINUATION ENVIRONMENT)))
    ))
```

There are some pieces missing from the present implementation. Were we to develop Scheme into a parallel programming language, we would have to provide facilities for communication between programs. At present, different Scheme programs that are running in parallel can communicate only through a shared global environment, and there are no facilities for reducing the contention that results when programs share environments. The syntaxer is an artifact of the CLI simulator's MacLisp implementation and does not address the issues involved in building real CM linkers and loaders. We have not thought about garbage

collection.

Almost everything we learned in the course of writing the interpreter is so obvious in retrospect as to be nearly invisible. (The most confusing aspect of the interpreter, the BIND protocol, arose naturally and was confusing only in retrospect.) There were three main stages in the program's development:

- The first version was written almost entirely in terms of the coding cliches described in the previous section.
- The second version introduced the FORKing trick described in the context of CONDITIONAL and so implemented the tail-recursion of the Scheme interpreter properly. Various optimizations to avoid unnecessary waiting by cells made most uses of the DEFCOMMAND and DEFMAKER cliches disappear.
- The third version made DEFINE expressions work properly and considerably cleaned up the interpreter's modularity by changing the implementation of environments from a simple list of binding pairs to the two-level frame-and-alist structure.

We have used the CL1 Scheme interpreter, in simulation, to run some substantial Scheme programs, including a simple relational database system (about 100 lines of code). The main thing we learned from these exercises is how much of our usual programming practice assumes a serial virtual machine. Programming in Scheme for a highly parallel virtual machine, we found that efficiency considerations often differentiated between ways of writing our programs that were equally efficient on serial machines. Although our programs ran correctly, we have little absolute idea how quickly they would be executed on a real CM. On the basis of the few rough calculations we can perform, we guess that the cross-over point between running N Scheme programs in parallel on the CM and in serial on a serial mainframe is reached with N in the thousands.

Some Connection Machine efficiency considerations

A working programmer constantly makes design decisions according to some model of what kinds of processes are efficient or inefficient on her machine. Our ideas about how to design CM languages and write logically correct programs in them are derived from bits and pieces of ideas about the corresponding problems on serial machines. Nothing, however, has prepared us to reason about the relative efficiencies of different logically correct CM programming methods. It's very hard. And lacking an actual CM, we have little empirical evidence on which to formulate ideas about efficiency considerations.

On one analysis, the central consideration is that the whole machine has only one instruction stream and so only one type of cell can be running during any given wall-clock tick. To a first approximation, then, in a program with a hundred different states 99 of them will be dormant at a time. But because the CM's controller is free to run the states in any order it likes, this can be a poor approximation in real programs. Compile-time and run-time analyses can determine what the most heavily populated states are likely to be at a given time. Since semantics of CL1 dictate that the outcome of a program is independent of the order in which the states are run, the CL1 programmer can't help the controller out. This is one less thing for the programmer to have to worry about and one less opportunity to use one's understanding of what happens when in designing an efficient program.

Such considerations might be entirely irrelevant if, as some of its designers expect, the CM turns out to spend almost all of its wall-clock time routing messages. In that case, the subtle statistical issues of router

congestion will predominate in reasoning about efficiency. If the programmer can't derive or intuit any models of these matters to guide her efficiency judgements, she's going to be in trouble.

A CM programmer pays not for the total amount of computation done by her program (as on a serial machine) but for the number of different things that must be done on each step. Consider, for example, an algorithm for locating the unique element of a linear list that satisfies some complicated predicate. On a serial machine it is best to test each element of the list in turn, proceeding to the next element only if the test fails. On the CM, one would be better off traversing the whole list putting each element into a state in which it is about to apply the predicate. Then all of the predicate tests can be run in parallel. Alternating predicate-test and next-element operations would allow only one predicate-test to be executed at a time.

Another important consideration in CM programming is that one often pays not for the average case of an algorithm, as on a serial machine, but for the worst case. Consider an algorithm in which every cell in the machine goes through some loop until some calculation converges. Since only one sort of thing can happen at a time, the code for the inside of the loop must be broadcast repeatedly until all one million cells' loops have converged. If a cell converges it must sit idle until all its siblings have converged too.

We have deliberately avoided discussing the more traditional considerations of SIMD machine programming, many of which concentrate on formulating one's algorithms so that the computation is as homogenous as possible. This is frequently possible in applications that are governed by differential equations, as in low-level vision, or their symbolic analogs, as in constraint networks. The applications in which we are interested, however, generally do not appear to enjoy this luxury.

What we learned

Because we had little *a priori* idea of what would make a good Connection Machine programming language, CL1 is inevitably little more than a minimal extrapolation from the archetypes of the culture of serial machine programming. Experience using CL1 has led us to a number of conclusions as to what the next language should look like.

CL1 has no mechanisms for explicitly assigning a type to a cell. Instead, the CL1 programmer typically uses the built-in state mechanism to give types to cells. A cell is considered to be of a certain type just in case its state is in the set of states that implement the type. A difficulty with this is that the CL1 language forces all of the states associated with a certain type to be collected into one place. For example, consider the code for the `CONS-NODE` cell type provided above. A cell placed in the state `CONS-NODE` performs the usual handshake with its owner and then goes into an infinite loop receiving commands and responding to them. This command loop understands two types of commands, `GET-CAR` and `GET-CDR`. The cell responds by transmitting the requested part back to the continuation provided in the command.

Suppose we were to use `CONS-NODE` cells to store a list of items in the conventional Lisp manner, linking through their `CDR-PART`s a series of `CONS-NODE`s whose `CAR-PART`s are the elements of the list. Given the definition above, it takes four messages to retrieve the second element of a list: two commands and two responses.

If we modify the definition of `CONS-NODE` by adding a new "method" (in the sense of current

object-oriented programming languages [8, 16]) to its command-type dispatch:

```
(DISPATCH COMMAND
  ...
  ((GET-CADR CONTINUATION)
   (TRANSMIT-WAIT CDR-PART (MAKE-COMMAND 'GET-CAR CONTINUATION)))
  ...
  )
```

then we can obtain the second element of a list in only three messages: a command of type GET-CADR from the requesting cell to the first cons, a GET-CAR command from the first cons to the second, and a third message from the second cons back to the requester.

Having to modify the code that implements the CONS-NODE type whenever a new operation such as GET-CADR is needed is a severe modularity violation. We would prefer to be able to write our methods wherever convenient, potentially in lexically distant locations. The CL1 compiler could achieve this by looking through the program and collecting together each set of related methods before beginning compilation of the lexical context in which they all must appear. But this is only a shallow solution to a deep problem.

To see why, suppose we additionally required that the CL1 compiler *automatically* generate additional methods whenever they can be useful. For example, the GET-CADR method we demonstrated above should ideally be generated whenever the programmer writes something like (GET-CAR (GET-CDR ...)) in her code; an analysis of the message passing should reveal to the compiler that two of the four messages involved can be compressed into a single message directly from the first cons to the second. Such optimizations as this require the compiler to consider all relevant entities simultaneously, but the CL1 compiler's view encompasses only one cell at a time. We would like the compiler to maintain a compile-time model of the joint behavior of *several* cells.

This requires the compiler to identify situations in which each of a group of cells is in a known state. It can do so when, for example, a command has been received by a CONS-NODE cell from a particular type of cdr-requesting cell. When this happens, the state of both cells is completely determined: the requesting cell must be waiting for a reply to its request, and the cons is starting to reply. The compiler would now be free to expand its viewpoint to include both cells, if only the language allowed the user to tell the compiler which states can send GET-CDR commands to CONS-NODEs.

A related shortcoming that can be traced to CL1's cellular view of the machine is the excessive amount of code devoted to message handling in most CL1 programs. CL1's support for message transmission cannot be made significantly more powerful because message transmission is a non-local phenomenon and CL1 maintains only a local model of the machine's behavior. Because the code implementing our communications protocols was often spread over several lexically distant locations, we found it difficult to define abstractions for them. The macros we did write came in cooperating sets: DEFMAKER, DEFOBJECT, and FORK all use the OWNER convention, and DEFCOMMAND and DEFOBJECT both use the command channel convention. These macros do not precisely abstract the conventions they use but rather particular, local, ways of using them. When we revised the code to increase parallelism and reduce message transmission, the macros usually failed to capture the resulting patterns.

For example, when the initialization of a cell involved more than the OWNER protocol, DEFOBJECT was no longer useful (and could not be extended to be useful). Likewise, macros defined with DEFCOMMAND failed in the presence of tail-recursion. Each of these macros expands into a form that waits on a result and returns it. When, as in the evaluation of the THEN or ELSE expressions of a CONDITIONAL, we intended the result from a command to be sent not to the originator of the command but to some other continuation, we had to revert to constructing the command explicitly.

It was exactly CL1's relative inability to allow users to define abstractions of message-passing cliches that defeated an attempt to implement the rule-based language Amord [7] in CL1.

Yet, the clearest indication of the underlying problem is how difficult CL1 code can be to read. When hand-simulating the execution of a CL1 program, one's finger must jump from page to page, following chains of causality that were real enough to the programmer but are only implicit in the code. This is graphically demonstrated by the code for the interface between OPERAND-CONSes and ALISTs, in which no indication is given as to which TRANSMIT corresponds to which INPUT.

These symptoms all reflect a single disorder. There are two kinds of causality vectors in CL1, state-transition and message-transmission, but CL1 explicitly represents only the first. Because of this, code must be grouped inconveniently, common patterns of causality are difficult to abstract, the behavior of the code is hard to reason about, and the compiler can have only a weak model of the computation. For the sake of both the user and the compiler, then, a programming language must make all causality explicit.

We are implementing a successor to CL1 that does not distinguish between state-transition and message-transmission, or equivalently, between objects and messages. By unifying these two kinds of causality into a single construct, the new language, called CGL (for Connection Graph Language) allows all causality to be made explicit. This proposal realizes the duality between message and recipient that can be seen in the protocol for creating and chaining together new ALIST cells. That protocol can be described either as a collection of processes passing an object around, or as a single process moving from object to object. We will report on CGL in [5].

To demonstrate how this unification can relieve some of the problems we experienced, consider the breakdown of our simple macros when we optimized our code. The way in which our proposal addresses this issue can best be understood by considering an analogy between a CM compiler and a Scheme compiler. In [12], Steele demonstrates how a compiler with a strong understanding of the simple semantics of Scheme can ease the macro writer's task by reliably optimizing the results of macroexpansion. The macro writer is freed to concentrate on the semantics of the macro. We want macros as simple as those defined by DEFMESSAGE and DEFOBJECT to express communications protocols without paying a runtime penalty. Thus we want a CM compiler to perform the kinds of optimizations we performed by hand on the results of expanding our simple macros. For this even to be theoretically possible requires that the compiler have a non-local model of the computation; for it to be reliable and efficient requires that that model be *simple*. We hope that our unification of state-transition and message-transmission will play the role that the lambda construct plays in Scheme.

The CL1 distinction between state-transition and message-transmission was derived from the CM architecture, which makes exactly that distinction. This correspondence determines an obvious assignment of

events in the virtual machine to events in the physical machine. Once we unify our two forms of causality, the compiler acquires considerably more freedom in deciding what quantities move about in the hardware and which stay fixed in processor cells. The `BIND` command, for example, moved four quantities in its fields down a chain of `OPERAND-CONS` cells, each of which has only two state variables. The compiler ought to have the option of keeping the processes embodied by propagation of `BIND` commands fixed and streaming chains of `OPERAND-CONS`es over them.

Acknowledgements

The Connection Machine was Danny Hillis' idea. David Chapman, David Christman, Carl Feynman, Brewster Kahle, Tom Knight, Bill Kornfeld, Glenn Kramer, Cliff Lasser, Charles Leiserson, Henry Lieberman Chris Lindblad, David Moon, Paul Rosenblum, Gerald Sussman, Jon Taft, and Richard Zippel have all contributed to our understanding of the problem of programming Connection Machines. Some of the ideas in this paper are part of Connection Machine culture and should be credited to them. Penny Berman provided important comments on a draft of this paper.

Bibliography

1. Abelson, Harold, and Sussman, Gerald Jay, course notes for MIT course 6.001, "Structure and Interpretation of Computer Programs", forthcoming.

2. Aho, Alfred V., and Jeffrey D. Ullman, "Principles of Compiler Design", Addison-Wesley (1977).

3. Backus, J., "Can Programming be Liberated from the von Neumann Style? A Functional Style and its Algebra of Programs", Communications of the ACM, Vol. 21 no. 8 (August 1978), 613-641.

4. Bawden, Alan, "CL1 Manual", MIT AI Working Paper 254 (Sept. 1983).

5. Bawden, Alan, "A Programming Language for Massively Parallel Computers", MS Thesis, Dept. of Electrical Engineering and Computer Science, MIT, September 1984.

6. Christman, David P., "Programming the Connection Machine", MS Thesis, Dept. of Electrical Engineering and Computer Science, MIT, January 1984.

7. de Kleer, Johan, Jon Doyle, Charles Rich, Guy L Steele Jr, and Gerald Jay Sussman, "AMORD: A Deductive Procedure System", MIT AI Memo 435 (Jan. 1978).

8. Goldberg, Adele, and David Robson, "Smalltalk-80: The Language and its Implementation", Addison-Wesley (1983).

9. Guzman, Adolfo, Miguel Gerzso, Kemer B. Norkin, S. Y. Vilenkin, "The Conversion via Software of a SIMD Processor into a MIMD Processor", Proc. IEEE Workshop on Computer Architecture for Pattern Analysis and Image Database Management, Oct. 1983.

10. Hewitt, Carl, "Viewing control structures as patterns of passing messages", AI Journal, Vol. 8 no. 3 (June 1977), 323-363.

11. Hillis, W. Daniel, "The Connection Machine (Computer Architecture for the New Wave)", MIT AI Memo 646 (Sept. 1981).

12. Steele, Guy Lewis Jr., "RABBIT: A Compiler for SCHEME (A Study in Compiler Optimization)", MIT AI-TR-474 (May 1978).

13. Steele, Guy Lewis Jr., and Gerald Jay Sussman, "LAMBDA: The Ultimate Imperative", MIT AI Memo 353 (March 1976).

14. Steele, Guy Lewis Jr., and Gerald Jay Sussman, "The Revised Report on SCHEME: A Dialect of Lisp", MIT AI Memo 452 (Jan. 1978).

15. Sussman, Gerald Jay, and Guy Lewis Steele Jr., "The Art of the Interpreter or, The Modularity Complex", MIT AI Memo 453 (May 1978).

16. Weinreb, Daniel, and David Moon, "Lisp Machine Manual", Symbolics Inc. (July 1981).