

PROCEEDINGS
OF THE
1985 INTERNATIONAL CONFERENCE
ON
PARALLEL PROCESSING

August 20-23, 1985

Douglas Degroot
Editor

Co-Sponsored by



Department of Electrical Engineering
PENN STATE UNIVERSITY
University Park, Pennsylvania

and the



IEEE Computer Society

In Cooperation with the



Association for Computing Machinery

IEEE
**COMPUTER
SOCIETY
PRESS** 

LAZY EVALUATION AND CANCELLATION OF COMPUTATIONS

David Hemmendinger

Department of Computer Science
Wright State University
Dayton, Ohio 45435

Abstract -- This describes a message-passing system for lazy evaluation of an FP-like language on a system of processors. A graph-reduction machine, it supports parallelism, non-determinism, and the cancelling of computations begun but no longer needed. Principal results are the message-passing apparatus, which provides a parallel remote procedure call semantics and the analysis of the cancellation of tasks.

The Language: FPL

Cancelling unneeded computations is a natural extension of lazy or demand-driven evaluation; e.g., a boolean function may cancel the call for one of its arguments if the other suffices. Lazy evaluation also permits a language to tolerate diverging computations--a function may return a value if the diverging argument is not needed, and then cancel the unfinished computation.

FPL differs from Backus's FP [1] in permitting sequences to be well-defined even when they hold undefined elements. These are function applications that have not returned values, either because they have not yet been evaluated, or because they are endless computations. Objects in FPL are numeric and symbolic atoms, sequences of objects, and the determinate error object, TOP. This object must be distinguished from BOTTOM, which denotes a function application that does not [yet] have a value; this cannot be detected, but plays a role in the semantic specification. In particular, a sequence has a definite length even if all of its elements are BOTTOM, and a selector may return any of them that is not BOTTOM. FPL functions force their arguments to the least degree that permits them to return a definite value. Computations are assumed to go in parallel whenever possible, so that an undefined element that is not needed will never prevent completion. The binary boolean functions are short-circuiting; if the first argument to be returned suffices, they return a value without waiting for the other (this is the principal source of cancellations).

The following definition of OR in terms of primitive type functions translates quite directly into an operational specification:

```
OR:x = if not pair(x) then TOP
      else for {i,j} = {1,2}
          if bool(x.i) then if x.i then T
              else if bool(x.j) then x.j else TOP
          else if bool(x.j) then if x.j then T
              else TOP
```

OR thus issues parallel demands for the two components of its argument, and a single true value completes the computation whether the other is boolean, non-boolean, or unknown. This definition is equivalent to one using McCarthy's "amb" function, but is operationally clearer.

Message-Passing in a Reduction Machine

A program in a graph reduction machine is a rooted digraph with function nodes and "apply" nodes that initially hold references to function and argument, and eventually hold the value of the function application (there may also be primitive value nodes). The program executes by using function definitions as rewrite rules to transform the graph until it is a single node holding the value of the computation. In the FPL machine, the apply node is an actor-like structure, called a cell, that exchanges messages with other cells and encapsulates all local state-changes that occur during function execution. It plays the role of a process in a conventional machine, and may be represented as a triple <program stack, workspace, <WAITERS, WORKERS, RESULTS>>, where the last element is a triple of message lists. WAITERS is a list of demanding cells, WORKERS, a list of cells to which demands have been sent, and RESULTS, a list of values received in response to demands.

The demand-driven evaluator begins evaluating a cell when it receives a demand, or schedules it for evaluation if no processor is available. If the argument is already evaluated the computation can complete; if not, the cell in turn issues demands and waits for the results. Once evaluated, a cell can respond immediately to subsequent demands. A cell that is not actively executing is in a state characterized by the contents of its WORKERS list; state transitions occur when a cell becomes a currently executing one. There are three states: idle, working, and suspended.

There are also three kinds of messages: 'eval', 'result', and 'cancel'. Initially and finally, a cell is in the idle state. If it receives an 'eval', it either responds without leaving the idle state, or stores the 'eval' in WAITERS, issues 'eval's to other cells, storing copies of these in WORKERS, and waits. It is in the working state, having distributed parts of the computation to other cells. When all the 'result' messages arrive, the cell reverts to idle.

A cell that acquires enough results to determine its value may issue a 'cancel' to all remaining cells listed in WORKERS. It then forgets them and becomes idle. Receiving a 'cancel' is more complicated. Since a cell can only receive a message when it is not actively executing, either the 'cancel' arrives too late, or the target cell has in turn sent 'eval's to others. In the latter case, the 'cancel' may have to be propagated. There are two cases. If the cancelling cell is not the only one listed in WAITERS, then the recipient merely deletes the corresponding demand from WAITERS. If WAITERS thereby becomes empty, then 'cancel's are sent in turn to all cells listed in WORKERS, and copies of these messages are stored in WORKERS in place of the 'eval's. The cell is suspended; this state is necessary

because the cell getting the 'cancel' is the root of a subgraph that may be known to others that have not yet issued demands. If these are sent later, the subgraph must resume work by converting the stored 'cancel's in WORKERS to 'eval's and sending them out again.

Implementation

The present implementation, written in an intermediate stack language and run on a PDP-11, divides the work between an evaluate routine that initiates demands and executes functions, and a message-handler. The latter routes messages and serves as a filter, placing messages in the appropriate lists of a cell and returning control to the evaluator when results are available. The intermediate language provides the data structure, a sequence-like object, that represents cells and all other objects. A reference-counting garbage collector manages storage. Messages have references to sender and receiver, and the copies stored in the message lists keep cells in existence as long as they are known to any part of the graph.

The design and implementation are robust. Since cells are multi-tasked and most of the work is done by starting up other cells, any cell that can finish its work will do so; an endless loop can only be an endless generation of new cells, which cancellation should prevent if they are unnecessary. The message-passing apparatus can readily be extended to multiple processors, and can tolerate lost messages. Lost 'cancel's will not generally hurt beyond slowing progress, and a lost 'eval' or 'result' can be handled by sending another 'eval' after a timeout. While messages have been assumed to arrive in the order sent, this is not necessary; a 'cancel' that arrives before the corresponding 'eval' can just be held pending arrival of the 'eval'.

This implementation is slow, partly because of the use of an intermediate language, and partly because messages must be sent for every small computation. Our plans include work with heuristics for packaging more work within a cell, so that the machine has eager as well as lazy components, and fewer messages are exchanged.

Analysis of Cancellation

A computation that is cancelled may be a well-behaved one that would terminate, or a divergent one. Because FPL is BOTTOM-tolerant, these ill-behaved functions cannot prevent the entire computation from finishing, but they can consume space and time if allowed to spawn new cells. There are thus two distinct issues: whether cancellation will save significantly when stopping a well-behaved function, and whether a runaway function can always be stopped eventually. In addition, we must be sure that cancellation never prevents a needed value from being produced.

Our message-passing, in which 'cancel's are passed from one cell to the next, insures that no cell will be suspended in the presence of outstanding demands, and that the offspring of a suspended cell will always be informed that they are suspended. Cancellation is therefore correct, and any unneeded and well-behaved computation will

get cancelled. Since a computation does most of its work by creating new cells to evaluate functions as they are demanded, and these are the last to receive the chained cancel messages, they may have finished before getting the cancellation. This method of cancelling is thus very slow.

In fact, if cells are assigned to processors when they get 'eval's, and messages are processed in the order sent, a 'cancel' will always arrive after its target has started evaluating and sent 'eval's to its offspring. If there is always a new generation, the 'cancel's will never quite catch up, and if two offspring are spawned at each step, unwanted but active cells grow indefinitely.

An obvious answer is to give priority to 'cancel' messages. This is easy, but if there are many processors, then 'cancel's may still not arrive in time to stop the next generation. A runaway computation will eventually glut the system with new cells, though, so that each processor will have a list of cells waiting to act on their 'eval' messages. Now the priority of the 'cancel' will let the processor handle it before the target cell starts work, and cancellation succeeds. We can be sure, therefore that divergent computations will get stopped.

While this propagation of 'cancel's through the graph assures correctness, it delays their reaching the leaves of the graph, which is where the activity takes place. An alternative would be to broadcast 'cancel's from a cell to all processors, saying "Don't let any of my progeny start," thereby stopping them all at once. There are two problems here. First, every cell must be identified as a descendant of its ancestor, which may cancel. This ancestor is not unique, however. Second, a cell may hold a second 'eval' and so the subgraph with it as root should not stop. While this root cell knows of the second demand, the remainder of the subgraph does not, so they must be restarted or not stopped in the first place.

A solution is to note that only some cells may cancel; let the cells representing them be "C-cells", and the others, "S-cells" (strict). Each C-cell names itself as ancestor when sending 'eval's. When one broadcasts a 'cancel', if it is not the first to demand a particular cell's value, there is no effect on that cell. If it is the first of two, then some cell knows of both demands, and so it reissues the demand in the name of the second demanding cell, thereby restarting the rest of the subgraph. Once again we have a chaining of messages, but now it is sparser.

There is another method. Since only the leaves of the graph must be directly reached by the broadcast 'cancel', let the cancellation ignore the other nodes. If these interior nodes are not needed by the higher ones, then once the cancellation of the leaves takes effect, the interior becomes garbage to be collected, since it is known neither to ancestors nor descendants. If a subgraph is still known by demanders, then it will be found by the garbage collector. Once found, it may be restarted.

Our proposal is thus to use the garbage collector to carry some messages. Let us use the highest C-cell in a subgraph as the ancestor of all below it. If it cancels, the garbage collector reclaims all lower cells not needed by another

cell. If another demand is found, the collector finds the name of the demanding cell and continues, now renaming the ancestor as this new cell. If a cell in the middle of the graph cancels, it does so in the name of the higher ancestor. This stops more than should be stopped, but the collector will restart what is needed.

In fact, this apparent disadvantage may be a virtue. A large subgraph gets stopped because of a 'cancel'; this is because some cell in the graph has a result. Since result messages may continue to be propagated upward, the suspension of all the leaves of the subgraph allow the result to be received and used faster. By giving priority to the return of the result as far up the graph as possible, more cancellations may then be possible. This method, then combines the breadth-first graph expansion of parallel lazy evaluation, with pruning of the graph whenever 'cancel's are issued.

Comparison with Related Work

Baker and Hewitt [2] propose a collector that periodically stops all processes (our cells) and does collection, restarting processes in its wake. This is similar to our method, except that our 'cancel's stop only a portion of the entire graph. Because they do not have 'cancel's, they use a scheduling algorithm to allot less processor time farther down the graph. Friedman and Wise [5] propose "sergeant" processes helping a main "colonel" process, so that when a node becomes useless the sergeants can be reassigned, but they do not show how to find these processes readily, nor have they implemented the method. Grit and Page [6] use a reference-counting collector like ours, running in parallel with the function evaluations. Unlike our graph, theirs does not handle common subexpressions unless they can be recognized at compile time. They point out that their method cannot guarantee to stop runaway tasks.

There appear to be no other discussions of the problem of cancelling tasks. As Treleaven, et al note [8], there are few reduction machines using lazy evaluation, and those that do, such as [3], [4], [7], do not deal with cancellation.

Future Research

It is difficult to know what portion of a graph will be stopped by this cancelling method. Since the highest C-cells are initially the ancestors of the rest of the graph, much depends on how close to the start these are, and how many there are in parallel. As the graph expands and contracts, ancestry changes. For that matter, there seems to be little information available about the efficacy of demand-driven evaluation. Our cancelling procedure is a kind of heuristic; we need to test it further to determine whether it, or perhaps our original method of sending explicit cancel messages down a chain, is worthwhile. It may turn out that the effort to guarantee that runaways be stopped is not worthwhile. On the other hand, an efficient method of doing that would also be efficient in terminating any well-behaved computation that is no longer needed.

Our present plans include a new implementation with several cancellation options available,

so that we may explore this issue. Since extreme laziness is expensive with small functions, we are also working on ways to have only larger computational units exchange messages, thereby simplifying interactions among processors. We are exploring heuristic measures of complexity that can help to decide when to assign a task to a new processor. In preparation for the multiprocessor implementation, we are also examining methods for distributed garbage collection, discussed in [9]. We are looking at ways of using compile-time program transformation to reduce the number of run time messages needed. A virtue of our message-passing model, in fact, is that reduction by program transformation and by run-time execution may be uniformly described by message-passing.

Acknowledgements

I wish to thank all the members of the FLITE group at Wright State, with whom this work was done.

References

- [1] J. Backus, "Can Programming Be 'Liberated' From the von Neumann Style?," Comm. of the ACM 21 (July, 1978), pp. 613-641.
- [2] H. G. Baker & C. Hewitt, "The Incremental Garbage Collection of Processes," SIGPLAN Notices 12 (August, 1977), pp. 55-59.
- [3] T. J. Clarke, P. J. Gladstone, & A.C. Norman, "SKIM—The S,K,I Reduction Machine," Proc. of the LISP-30 Conference, Stanford (1980), pp. 128-135.
- [4] J. Darlington & M. Reeve, "ALICE: a Multiprocessor Reduction Machine for the Parallel Evaluation of Applicative Languages," ACM Conf. on Functional Programming Languages and Computer Architecture (1980), pp. 65-74.
- [5] D. Friedman & D. S. Wise, "Aspects of Applicative Programming for Parallel Processing," IEEE Trans. on Computers C-27 (April, 1978), pp. 289-296.
- [6] D. H. Grit & R. L. Page, "Deleting Irrelevant Tasks in an Expression-oriented Multiprocessor System," ACM Trans. on Programming Languages and Systems 3 (Jan., 1981), pp. 49-59.
- [7] R. M. Keller, G. Lindstrom, S. Patil, "A Loosely Coupled Applicative Multi-processing System," Proc. of the National Computer Conference, AFIPS, (1979), pp. 861-870.
- [8] P. Treleaven, D. R. Brownbridge, R. P. Hopkins, "Data-driven and Demand-driven Computer Architecture," ACM Computing Surveys 14 (March, 1982), pp. 93-143.
- [9] K. A-H. M. Ali, Object-Oriented Storage Management and Garbage Collection in Distributed Processing Systems, Dept. of Computer Systems, Royal Institute of Technology, Stockholm, TRITA-CS-8406 (Dec., 1984), 225 pp.