Rod, I thought this might be of use, Kevin.

ML Progress Report

K. Mitchell.
13.9.83.

The purpose of this note is to bring people up-to-date with the state of the current ML implementation and to outline possible areas for future work.

The ML system can be split into

i) The scanner/parser
ii) The type checker
iii) The compiler
iv) The assembler
v) The runtime system (i.e. abstract machine interpreter)

With the exception of v) which is currently written in VAX assembler, the rest of the system has been converted from Pascal to ML. Some of the modules are very close to the original Pascal code (e.g. the type checker) whereas other modules have been completely rewritten (e.g. the parser). As far as we are aware, all of the translated modules are now working except for the parser which requires some additional work on error recovery and some more testing.

Because the whole of the system now resides within the ML heap instead of growing within the Pascal heap, the speed of garbage collection has become a very important factor in the

overall speed of the system. A typical page of ML code may require two or three garbage collections while compiling. The interactive response is normally quite acceptable.

There are two versions of the system currently available. The first provides an environment very similar to the Pascal version of ML (i.e. a few library functions such as append are known). The second version is more like a typical lisp system where all the top-level functions making up the system are accessible through the system's environment. For example, consider the 'debug' function previously hardwired into the pascal system. This has been replaced by a number of boolean ref variables that are not only accessible to the user as part of his environment but are also the variables that control the main loop of the system.

We have built in the ability to save and reload arbitrary ML objects (including circular ones) and can also save and restore a complete ML state. We have a version of the system that boots itself off of one of the saved states thus requiring no pascal code at all within the system.

To give some idea of the size of code currently being produced, the saved state of the simple system is about 200 Kbyte and the size of the self-referential version is about 300 Kbyte. [While there are various technical reasons why the self-referential version should be larger, the current increase seems

to be rather excessive and is one of the world's great unsolved mysteries that needs to be investigated!) This compares with the 250 Kbytes taken by the saved image of the pascal version (where no types and very few names are saved at runtime). The system is currently running with a 1.5 Mbyte heap but because a copying garbage collector is currently used the system needs two heaps (although only one is used at any instant in time) and approximately 4 Mbytes in order to run comfortably.

During the course of the translation we have taken the opportunity to incorporate a number of optimisations, the most important of these being the removal of a lot of the unnecessary argument pairing and destruction that went on in the original system.

The above work was carried out by Alan, John and myself. Over the last six weeks, in parallel with the above work, Nick and Prasad have been working on the system as well. Nick has been reimplementing the abstract machine in C as an 'easy' way of porting the system to the APM or a PERQ. By implementing a few critical sections of the abstract machine in assembler it may be possible to use the C version as the standard interpreter on the VAX as well, but it is too early to know if this is a realistic aim. Nick has currently got most of the interpreter rewritten in C but a lot of testing and tuning still needs to be done.

Prasad has been working on the provision of a trace and break package for the system. He has implemented a coroutine package for the system which allows the user program to be monitored by a 'big brother' process that can examine the user process' current return stack, trap stack etc. The two processes can share the same top-level environment so a limited amount of interaction is also possible between the debugger and its victim. Prasad is now looking at how to implement a trace package (similar to the one provided in DEC10 ML) as a complement to the break package.

The Future.

There are many possible directions for future development. Off the top of my head I can think of at least the following (in no order of importance)

- Sharable images (for student teaching)
- Reducing size/increasing speed
- Porting system onto APMs + FERQs.
- Standard ML
- Support for LCF.

I would like to make some points about a few of the above items (in no particular order as many of the points affect a number of the possible directions).

1) The previous pascal version allowed most of the system to be shared by many users; this was obviously a great help when a lot of students were trying to use the system. We now have the situation where the code for the system and the code for a user's program both reside in the same heap. This not only prevents us from sharing the system code but also means that we have to garbage collect the system code again and again, even though it is never changed (well, at least most of the time it isn't!). Some form of partitioned heap is required.

2) The system is currently quite large which will cause problems when we try to implement it on other machines. There are three obvious ways of reducing the size of the system.

   i) Replace the copying collector by some other garbage collector that only needs one heap space.

   ii) Make the rest of the system produce less temporary data so that the system can run in a smaller heap.

   iii) Make the code more compact.

How small does the system need to be? ML as it currently stands should run on a Perq if virtual memory is used. However there is no hardware support for virtual memory so it will probably run quite slowly (John may have some figures on this). To avoid the use of virtual memory the system plus UNIX would have to run in under 1 Mbyte (i.e. a heap of at most 756 Kbyte). To achieve this we would not only have

to change the garbage collector but also halve the amount of space required by the rest of the system. Although we have already identified some space inefficiencies in the system, it is extremely doubtful whether any of these could halve the size of the system (unless there is a bug somewhere consuming vast amounts of store).

The other machine that we are likely to want to run ML on is the APM. Here the problem is not that we can't run ML in physical memory but that it is probably too expensive to do so. However in this case we must trade off how much effort will be required to make ML much smaller against the likelihood of virtual memory appearing on the APM. An APM with virtual memory (hardware support chips already exist - the 68020 should soon be available) and 2 Mbyte of physical memory would easily be able to support the existing system (remember that, except when in the middle of a garbage collection, under 2 Mbyte of memory would be needed so very little paging should occur). We should find out when or if virtual memory is going to appear on an APM.

If we wish to replace the garbage collector then apart from the fact that we shouldn't underestimate the time taken to implement the new collector (the time spent coding it up is only the tip of an iceberg!), the new collector must still allow some form of partioned heap. Furthermore we currently implement the importing and exporting of objects (including the bootstrap

states) by using part of the garbage collector; any replacement must involve new ways of doing this as well.

3) A lot more information needs to be collected from the system before it can be effectively speeded up or decreased in size. None of us have anything but a crude idea of where the time + space are being consumed. There are a few obvious data structure optimisations that could be made but until we have more information on the type of junk being created, and where, such optimisations might require a lot of work for very little gain.

4) We should decide when, or if, we are going to move the implementation to standard ML, and what priority this move should have. If Dan is thinking of writing LCF on top of ML, how would this change affect him? Finally, can we break the standard down into sections and do a bit at a time or are some things so intertwined that the transition has to be done all at once?

While parts of the pascal system were still waiting to be translated into ML there was a great incentive to keep going and the project seemed to develop a lot of momentum. However, since the main translation effort is complete, I sense that the momentum has been lost to a great extent (probably I am most to blame for this as i've stopped spending twelve hours a day at the terminal!). To regain some of this momentum I think that it is important to decide on a clear set of concrete short-term aims as soon as possible.