CONTROVERSY

Portability – A No Longer Solved Problem

Stuart Feldman
Bellcore
W. Morven Gentleman
National Research Council of Canada

ABSTRACT: Of Man's First Disobedience, ... *or* Not long ago, the problem of software portability seemed on the way to solution. Now, the need and demand for portability is enormously greater, but solution seems much further away. Reasons include stricter requirements for portability, different expectations by and of the people doing the work, and a far wider range of environments in which software must function. This paper describes how this came about and modern approaches to the problem.

1. The Innocents Abroad[1]

Program portability is an old goal. There have been many notable successes, but there has been a major shift in expectations and satisfaction in the past decade. The first efforts were in the areas of mathematical and systems software, and a great deal of

1. All the section headings refer to actual works; no prize is offered to those who identify them. –PHS

research led to good practical solutions involving clever software and concentrated activity. Thus, we find ourselves in the paradoxical situation of having better techniques and knowledge but in more trouble and with more grumpy customers.

We contend that almost any program can be made portable, usually at acceptable production cost and execution effectiveness. There are many impressive examples. Even embedded systems with exotic peripherals and real-time constraints, have been successfully ported. However, the scale and scope of the problem have increased dramatically, and techniques need to be extended and upgraded to meet current demands.

In the old days it was rarely possible to move a system without change but it was important to avoid reinventing the whole thing. The idea that something was either portable or not soon broadened to a continuous measure of how portable an object was, depending on how much easier it was to move than to recreate from scratch. It is technically possible to create a completely portable version of almost any program since it is possible to reduce it to a Turing machine simulation, but the cost of such an approach is far too high. With careful study, computer scientists found some significant areas in which high degrees of portability were achieved at seemingly negligible cost. As a result of this achievement in limited areas, people are starting to demand absolute portability again, and in a far richer world.

As the cost of hardware drops, a growing fraction of "operators" are people who make use of computers, but are not knowledgeable about them, have no desire to delve into their mysteries, and do not want to spend time or thought in installing software.

In the past, moving a program was viewed as an extraordinary event. It is now the norm for any valuable code. Writing a program over from scratch is possible but also expensive, and the prospect is truly boring if it has to be done more than once. Many programs evolve radically, and the sequence of changes can usefully be described as a port from one machine to itself. If there are already versions for a number of environments, the functional changes then need to appear in all the environments, so there may very well be a sequence of parallel ports. This task can be a nightmare if not controlled carefully, supported by a

computer, and planned in advance. Conversely, significant economies are likely if one assumes that many ports will be done, and that some will be done in parallel. Forethought does indeed sometimes pay.

Of course, various panaceas have been offered. Language standardization has been presented as a necessary and sufficient solution to the software portability problem. (A major selling point of Ada was that programs would almost automatically be highly portable.) Not so. It is often possible to cover language inconsistencies, once they are known, by preprocessing and postprocessing tricks. The deeper problems are likely to lie in the ineffable interface or the unknowable configuration description.

Almost all successful portability efforts have been based on defining an ideal system to which each ported version is the best possible approximation given certain constraints. There are two basic approaches, the "Least Common Multiple," in which the model encompasses all the features that one anticipates encountering, and the individual ports are subsets of the ideal, and the "Greatest Common Divisor" approach, in which the ideal includes the minimal functionality, and the implementations include some adornments. The GCD approach is likely to be easier to understand, since the abstraction is simple, but the results may not be very satisfactory. More imagination and discipline are required to produce a sensible LCM model, but the approach is likely to provide better performance over a wide variety of targets since the model includes all the relevant aspects. It can be expensive to do an LCM port, but if appropriate tools are available, it can be straightforward to simulate complex ideas on a restricted machine. (For example, it is easy to implement an interpreter for the standard Pascal intermediate language PCODE and it is easy to translate Pascal into PCODE, but it can be very hard to optimize clever address expressions and to handle other such horrors. A more general intermediate is more laborious to implement, but that work can be mechanized, and it is then easier to do the hard job of generating good code on a wider variety of machines.)

2. *For Whom the Bell Tolls*

The basic reasons for wanting portable systems remain the same: portability increases the range of environments in which a piece of work can be applied with reasonable cost. The intellectual or financial costs of developing a serious piece of software can be amortized over a broader base, so a wider variety of jobs is worth doing and the rewards for the more obvious items increase. The maintenance costs can frequently be spread also, since many bugs and functional enhancements apply to all versions rather than being target-specific. The time to deliver to a market can be reduced dramatically by applying portability techniques: it is often the case that writing a program so that it can be ported has little effect on the time it takes to write the first version, but later versions can appear almost instantly. Some jobs simply would not be attempted if they could not be ported cheaply, since they may require some very scarce resource (typically, human expertise) to do at all; if an opportunity exists to do something once, it may be unrealistic to expect to do the job twice. The effective lifetime of a portable piece of software increases, since it is possible to transport it temporally to more modern pieces of equipment, not just to spatially separated ones.

The importance of portability is quite generally recognized. A recent international committee with the romantic name ISO/IEC JTC-1/TSG-1 has as its purpose making a recommendation to its parent committee what new international standards are needed, and what existing standards may need adjustment, in order to improve the portability of application programs.

Portability has advantages to a wide variety of people and organizations. In addition to the general considerations above, there are special benefits to people in particular roles:

- Integrated software system vendors and even hardware vendors are particularly interested in the ability to change substrate (computing environment) while preserving the surface level products. They can keep customers who are interested in a particular operating system or in a particular turnkey application over a long period of time and a wide spectrum of technologies.

- Independent (third-party) software vendors are especially interested in the ability to increase their market size by offering their product on a variety of platforms. They can do this with low redevelopment cost (whether measured in terms of money, time, or boredom) and without retraining their support staff.
- System integrators can offer their services in a wide range of platforms and for a wider range of components to integrate. They also have a business opportunity of evaluating, porting, and adapting software.
- Developers involved in geographically or corporately dispersed activity can share ideas and results over a wide range. This permits achieving critical mass even if individual organizations are small and resources are scarce.
- Educators care about portability because it increases the life-time of the skills they are imparting, permits students to get experience on a variety of equipment, and to have some commonality (e.g., programming language and editor) between courses.
- Naive end users are in favor of portable programs because they do not need re-training when hardware changes. The ability to exchange information with others without understanding fine distinctions is also of very high value.
- Managers and official decision makers are strongly in favor of portability because it increase the choice of platforms and facilitates phased upgrades (it is not necessary to change everything at once). It avoids illegal favoritism, increases competition in the marketplace, and leads to lower prices.
- Sophisticated individual users are likely to be pushing limits of their systems and are prepared to work at a low (code) level, but like the ability to choose from a variety of products to get around limitations or restrictions and enjoy access to rarely needed but exceedingly important tools.

A more general value of portability involves interactions with other tools. If a particular program is found in a variety of contexts, and other programs are similarly (but independently) ported, then the combination can be considered as a unit by a

user. If one can assume that a particular system will be available, then it is reasonable to write complicated programs that depend on interacting with it. The class of objects that are manipulated provides a level of abstraction that can be used in other programs, with full expectation that they can be handled properly in another environment. This idea applies to editors, screen managers, compilers, and a wide variety of other complicated services.

Personal mobility is also enhanced, since one need not recreate one's intellectual environment on a trip or after a career change. (Most but not all consider an increase in personal portability to be a social good. Consider recent events in Eastern Europe.)

Certain concepts become standard, and need not be defined and described each time they are wanted. Terminology can become widespread even when details change. (There are many ways to define and implement a pipe or to write a `for` loop, but no computer scientist will ask what you mean if you use the terms. Everyone knows what a byte or a floating point number is, without an exact definition.)

3. Economics

There are few hard data on portability (or almost any other area in software engineering). In particular, it is not easy to estimate the cost of creating a program or system to be portable, nor are there any uniform rules to apply to the costs of using a portable rather than a bespoke system.

Experience seems to indicate that the costs of doing a system in a portable manner are usually quite small. It may take intellectual effort and discipline, but the measurable expenses are probably a small percentage of the original design and implementation. The major expense comes from the need to test and package the numerous versions of a successful product.

The efficiency penalties incurred when writing a program in a general fashion can be quite hard to estimate, and effort estimates can be thrown off entirely if large amounts of ancillary software must come with the software being moved. It is not sufficient to move a subroutine or even a program – an entire world may need

to be moved. Part of the scaffolding used in developing the system may need to migrate with the main software in a well engineered system; the trickiest parts of a port can involve the initialization and general support mechanisms.

Nonetheless, as the complexity of the task grows and the breadth of environments to reach widens, it is necessary to ask if the goal of portability is still worthwhile and if it is worth paying the sometimes significant costs.

4. *Paradiso*

Once upon a time, it seemed that the problem of portability, at least for numerically based programs, was solved. Excellent mathematical libraries were available that worked well over a wide range of machines. They could be made available for a new range of machine in times ranging from days to a couple of months, with strong certainty that the effort would result in good software. A new language, Fortran 77, promised to bring order out of chaos and to present a safe model for scientific programming. The unpleasant jungle of floating point systems was being tamed by models to describe the well-behaved parts of most systems and also by the rapid spread of a well-designed and accepted (IEEE) Standard. It seemed possible to make programs function nicely over a wide range of input and output devices, including fixed-width cards, variable width printers, hard-copy and soft-copy character terminals. Certain systems, UNIX for example, seemed to be finding use on a wide variety of machines not originally designed to be a home. This trend would inevitably lead to uniform command interfaces and make life even smoother for users. The issue was so well understood that valuable books such as the conference proceedings edited by Cowell [1977], the collection of articles resulting from a course by Brown [1977], and the monograph by Wallis [1982] were published long ago. These books contain descriptions of successful techniques, accurate information, and insightful articles. Lesser books have since been written ([Dahlstrand 1984; Lecarme & Gart 1986; Henderson 1988]).

5. *Paradise Lost*

Over the last few years, it has been clear that either we have lost our way or were in a Fool's Paradise. It is not so easy to port large numerical codes intelligently from a scalar processor to a parallel machine. This was known long ago, but now many more people care about the answers because they have vector or array machines but lack staff to overcome the problems they cause. Fortran portability problems were addressed and answered quite satisfactorily, just in time for computer scientists to lose interest in the language and to press the claims of a broad spectrum of languages, many with terrible floating point properties. The UNIX system has continued to spread, but has become fuzzy in the process. A raft of versions are in common use (System III, System V, 4.2BSD, 4.3BSD, 4.0, lots of brand-name systems), various standards activities are underway (POSIX etc.), and some implementations are unfaithful. A user would be well advised to stick to a tiny subset of UNIX if faced with a new system, just as he kept within the uncomfortable PFORT subset of Fortran in the old days. So, have we carefully unlearned everything we knew?

6. *Something Happened*

The real problem is that we have become greedy, the world has become more complicated, and change has accelerated. The Golden Age involved a restricted range of hardware and software choices, and models had finally evolved that encased most of the uninteresting variations. Computers were still relatively rare birds, the number of providers of fundamental software was small, and certain users had large leverage on the vendors. Many of the successes involves mathematical and scientific applications, as witnessed by the very large fraction of Wallis's book that refers to such issues and examples. The other major class of successes described in the literature has been systems software, in particular operating systems and compilers. These are of course extremely important, the techniques have been very impressive (portable code generators at first appear a contradiction in terms), and the

result is tools essential for porting other programs, but one must remember that the goal for most users is *portable applications* not *portable tools*.

Now, there are orders of magnitude more providers of systems, and any individual buyer can have effect on but a few. Users no longer buy all their hardware and software from a single vendor. For example, UNIX-like systems abound; more than a hundred are available that use the 680x0, but they are not precisely identical – most would have no market if they were! The properties of Fortran that made it suitable for use in libraries also make it uninteresting to the vanguard, since the roots of the language are almost primeval, and the standardization that makes it a suitable portability vehicle also guarantees at best glacial progress. Whole language cultures can flourish and disappear between revisions of an international standard. We now want to transport more variegated items, not just number-crunching processes, but whole systems with complex data and display requirements.

Some of the basic assumptions underlying portability experience have been falsified. It used to be a rare and significant event when a product was to be ported, and the move would be done by a dedicated (in one sense or another) professional who knew a lot about the computing environment. In the world of microcomputers, most owners and users are (intentionally) unskilled in the mysteries of the machine, and see it as a vehicle for their other tasks, not as an object of inherent interest, and do not wish to spend time or money on installing or moving software. A piece of software is likely to be moved by a casual user who has no desire to follow a long set of steps and to make clever changes to the installation or operating instructions based on his (nonexistent) knowledge of his environment. When machines are connected by a network, users often expect to port software on demand, and to move software frequently and to perform computations symbiotically on several machines. Such a user is unlikely to give deep thought to the characteristics of the different environments of the connected machines, and expects the port to take seconds or minutes, not days. Such a user is likely to get quite annoyed at seemingly trivial and usually invisible problems like byte order; two's complement, one's complement, or sign-and-magnitude

arithmetic; absolute rather than relative memory locations in pointers; and representation of floating point numbers. He is not likely to be very sympathetic about different ways of storing information in files, or in changing names of global variables or data references. This lack of dedication comes at a time when the breadth of detail has increased alarmingly.

7. *Enemies of Promise*

Our systems grow ever more complex and assume more about their environment. A typical scientific program no longer does a mathematical computation and terminates. There is likely to be some sort of growing data base, complex input specification, graphical output, and interaction with other and very different programs. The models that could explain variation among Fortran processors and floating point units do not stretch well to cover cascades of languages, processors, and peripherals provided by multitudinous designers and vendors.

The following paragraphs describe some of the current degrees of variation that we see in the systems to which we might want to move our software. The naive user (ourselves on bad days) will assume that the distinctions are all beneath notice and for someone else to worry about.

7.1 *One Two Three ... Infinity* (Parallel and Distributed Computing Models)

For years we could assume a basic single-stream von Neumann model of computation, and this bias was implicit. The advent of the cheap microprocessor means that multi-processor configurations will proliferate. (Input/output devices often have an embedded microprocessor.) At the other end of the economic spectrum, the only way known to get enormous throughput on numerical calculations is through highly parallel machines. Array co-processors are quite inexpensive for their measured FLOPS rates, and Class VI supercomputers are no longer scarce. Though we dare not ignore the computational models that these machines

present, we do not have any simple description (and certainly no parametrization) of this space. Further to enrich our lives and complicate our descriptions, future machines can be expected to be inhomogeneous congeries with specialized components for graphics, floating point, and other well-defined and highly restricted uses.

7.2 Speak, Memory
(Storage Sizes)

Changing technology presents other opportunities. Memory costs are dropping fast enough that machines with enormous memories are being delivered. High-end machines now demand large stores. Memories of 64MB or more are common on fast minicomputers and mainframes; some mainframes cannot be purchased with less than that amount of memory per processor. Some supercomputers are offered with gigabytes of memory. Virtual memory systems are common but by no means universal. An algorithm that is clever on a machine with 64KB of memory is nonsensical on one with 1MB of memory. Certain programs that are right for a 1MB machine will probably prove outlandish for one with 100MB. (For a concrete example, consider the Fast Fourier Transform. When space is at a premium, the computation can be done in place, but at the cost of requiring a "digit reversal" transposing of the data which is intricate and, especially for mixed radix problems, time-consuming. If enough memory is available, orderings of the data at intermediate stages of the algorithm become possible that can, for example, maximize vector lengths, thus improving performance on vector machines. If lots more memory is available, redundant storage of trigonometric values can be used to further speed the computation.)

It is becoming more common to provide important functions in ROM, PROM, or on microcode floppies. The lifetime and change cycle for such pieces of not-so-soft-ware is very different from that of ordinary user programs, and may involve peculiar linkage and error signal conventions.

7.3 The Sign of Four
(Number Representation)

The success of the IEEE Standard should not blind us to the continued life of IBM-hex and Cray-style floating point numbers. The IEEE "Standard" is actually a floating point standard generator: it contains so many implementation options (which subset of the formats to support, parameters of the extended types if any, meanings of NaNs, responses to exceptions) that simple software cannot be written to work over unspecified IEEE Standard arithmetic.

Integer representations can also cause problems. Most machines have word lengths that are powers of 2, but the default integer length is either 16 or 32 depending on hardware architecture or compiler choice. Very interesting bugs can result from unintended wraparound at 65536. (We may be starting the cycle again with longer addresses and 64 bit integers.)

Even if the binary representation of quantities is agreed, the order of bytes is unlikely ever to be settled. Big-Endians and Little-Endians see no reason to change their ways to accommodate the others. Combining that freedom with choices of floating point representations produces the need for messy bit-twiddling programs which must know the exact format of the data stream on which they are operating. Moving data between machines then becomes even more difficult, especially if these delicate programs to fiddle bits must also be moved and be transformed.

7.4 The Time Machine
(Time Representation)

Representations of time are numerous and contradictory. Julian seconds are OK but cumbersome. Virtually every other representation is a mess. Decoding requires knowing abbreviations and full forms of month names in a variety of languages, rules for leap seconds, leap days, and leap months in variously formulated leap years, arithmetic in Roman and Arabic numerals, and various different encodings for negative numbers (BC).

Even with the ability to decode various strings and to produce similar ones, there are actual dangerous ambiguities. If an

American is sent a date by a European, the best thing to do is ask – you cannot tell whether the date is in European order (Europeans typically use *d-m-y* order) or in American *(m-d-y)* order, or if a European has politely reversed the order for American tastes. Otherwise, if you arranged a meeting on 6-7-89, should you have shown up on the seventh of June or the sixth of July?

7.5 *Gruesome Alphabets* (Character Representations)

EBCDIC is still with us, as are some 6-bit fossils. ASCII is common, but there are many freedoms for choices of national characters. The C language uses all but one of the 95 printing characters in the set, so even a simple printout of a C program is likely to look strange outside the U.S. Some terminals use the printable characters to control special functions ("in-band signaling"), so the result of copying a document to the screen may produce truly astonishing and nasty results. There is no agreement at all on representing more general graphics with multiple fonts and larger character sets. Even issues of whether to use more bits of characters or shift sequences are not settled. The X3.41 extensions, with multiple bytes per character, shifts, and loadable character sets, are a messy solution of the wrong problem.

Related important problems arise when representing words. The idea of a collating sequence (sorting sequence of individual characters) is simply not sufficient to sort words. Depending on context and use. "Mc" and "Mac" alphabetize the same, "St." (Saint) alphabetizes before Sir at the beginning of a name but after it at the end of a name (Street). Rules in other languages are even more difficult.

7.6 *Information Please* (Complex Data)

Even if we can reach some agreement on low-level number and character representation, the handling of more complex data structures is very hard. Keeping consistent representations of aggregates, relationships, and pointers can be a nightmare. Efficient mappings between different databases (relational, entity-relation,

hierarchical, object) are not generally possible. Transferring hypermedia documents among systems can be impossible (if some of the media are not available on the target) or just very hard.

Luckily, there are some de facto standard representations that can help. PostScript provides a usable intermediate representation for complex text (though color and images continue to be real problems.) Various representations of sounds are also broadly usable.

There are success stories in the area of commercial computing: spreadsheet and desktop publishing programs can exchange information, formatting, and even pictures across machine ranges and software vendors.

7.7 *Flatland*
(Higher-Dimensional I/O)

Most of us now use two-dimensional output devices (graphics screens), and many have quite fine-grained resolution. There is no standard representation of the abilities of such devices, nor how to use them to present pictures or even line drawings. There are crude approaches to describing broad ranges of terminals (the UNIX *termcap* facility, X3.64), but these do not really solve the problem for the coming range of devices. A multi-window capability can emulate a variety of output devices on a single screen, but it cannot satisfy varying needs for resolution, color, or increase the amount of screen area. The increasing availability of implementations of X.11 at least provide a common model, despite dissatisfactions.

A single computer will probably have several output units; even today it is not unknown for a user to have a simple terminal, a low-resolution color screen, and a high-resolution monochrome display in close proximity. This is a temporary solution for the problem that inexpensive computers often have more attractive output abilities (color, pictures, motion, voice and music) than do most expensive ("professional") equipment. (The latest workstations finally provide acoustic output, but not always input, and, at a price, high resolution color.) Quality printing used to be the preserve of specialized firms, then of large organizations; but printing suitable for publication can now be produced on laser

printers that are trending toward the hobbyist price range. We do not have any good way to control the mapping of abilities across this broad range of resolutions and software. If mixed or proportionally spaced fonts are permitted, simple techniques for locating or describing pieces of text become senseless. How long till we have three-dimensional output (stereo screens, milling machines)?

One of the safe assumptions in portability efforts has been that input would be a linear string of characters from a keyboard, and that input and output would not interact. The exact representation and lengths of lines cause discomfort, but programs could be written easily to circumvent the problems. Pointing devices (mice, light pens, touchscreens, and so forth) are now ubiquitous, but we have no generally accepted model for their interaction with programs. Dynamic reshaping of windows, following cursor movements, and more exotic interactions are both hard to program and describe at present. Menus are a venerable idea, but there is no standard interface software for producing, presenting, and signaling them. Each window system has its own idea of a good way to describe and present menus. How are we to cover this deficiency in portable programs and still permit use of hardware to which users are accustomed?

Perhaps the real problem in trying to develop device-independent programs is that there are no device-independent humans.

7.8 The Sound and the Fury
(Acoustic I/O)

Until recently, only inexpensive home computers and large business-oriented machines came with sound production devices, the former for sound effects in games, the latter for voice response to salesmen. There are now a number of devices for producing sound. There are several standards: MIDI, digital sound (CD encoding), μ-law, analog signals. In addition, there are proprietary interfaces for speech generation.

There are no generally followed standards for acoustic input.

7.9 Wired
(Communications)

Computers communicate, both with humans and with other automata. Communications systems usually operate in terms of protocols, which are defined more or less exactly. When the definitions are loose, it becomes very difficult to design software or hardware to send data to another site. If the definitions are too tight, they can restrict progress or make some important functions impossible. The X3, X28, and X29 communications standards for start/stop terminals seem benign, but effectively demand half duplex (alternating one-way rather than simultaneous two-way) communication over certain classes of network. Obviously, UNIX systems will work badly over such paths.

Networks have different packet formats, header contents, and ways of identifying addressees. The situation is sufficiently anarchic that there is not necessarily a unique parse of a mixed UUCP and ARPANET (R.I.P.) address.

7.10 Dead Souls
(File Systems)

Almost every computer comes with some form of persistent modifiable memory, and ways of storing and retrieving data in that memory. Almost all other aspects of the file or database system are variable. The naming scheme is sometime flat, sometimes tree-oriented, sometimes unconstrained graph-oriented, sometimes content-addressable. The names often have awful restrictions (choice of characters, total length of name, number of components, or length of individual components) which can make portability peculiarly irksome. The name itself often carries undesirable information about file location or purpose; these restrictions are invisible and highly idiosyncratic. The form of the data that may be saved or the file system's assumptions about the content can make it impossible to write certain simple applications in a portable way. Different systems store ancillary information with files: modification times, generation or version numbers, enforced data layout, or impregnable protection schemes.

The only way around most of these problems is to interpose a program that models a desired and well-defined file system between the application and the given system, then to try to implement that model on each system. The implementation process is likely to be tedious, and the result to be bulky, erroneous, and slow.

7.11 Also Sprach Zarathustra (Languages)

Our old rock, Fortran as lingua franca of scientific computation, has split. Now that Fortran 77 is common, it is suffering the irregular implementations of most languages. The quirks of the major implementations are legion, and the compilers have many poorly tested releases. No PFORT77 has been found in which it is safe to program. So it is probably time to start worrying about PFORT-88X.

Fortran is no longer a favored language, and a compiler is unlikely to be found on most low-cost systems. Even if a compiler is available, the user may not want to learn that unfortunate language, and will insist on writing his code in something more suited to human production. But other languages are in an equally bad state of standardization: (Do you have K&R C, ANSI-C, or a mongrel? How Common is your LISP?)

It is common for programs written in different languages to interact with each other. Even if a user succeeds in writing a monoglot program, the interface to the operating system will almost certainly involve a language shift (mixed language programming), since these interfaces are frequently governed by the necessities of hardware architecture or speed of assembly language coding rather than for meeting the needs of a general language.

People often advocate standards as a solution for problems, but there are many traps. A standard will either stifle innovation or force it underground. A standard that is too permissive will prevent gross bugs, but leave lots of subtle incompatibilities.

7.12 Cultural Literacy
(System Conventions)

A program must not only provide a basic function, it must fit in nicely in its new environment. It is necessary that it have the right "look and feel" and so must be consistent with the expected human interface and the way other programs in its new neighborhood will react. If everyone else uses menus, you should too. If terse responses are the norm, reticence is a virtue. (If the system mimics punch cards, should you put sequence numbers on your lines?)

7.13 The Whole Earth Catalog
(System Descriptions)

We must cover both finer and broader distinctions if our systems are to seem portable. In one sense we have returned to the earliest days of computing, when every machine was hand-crafted and very different from the others. A purchaser of a workstation or personal computer is likely to pick a configuration or get a set of boards that is detectably different from most others. He will also buy software components from a multitude of vendors, and these too will differ from those used by his friends. These differences may be at a very fine level (choice of processor or memory level, particular ROM or co-processor), but the software ought to conform gracefully to these alternatives. We are also accustomed to being able to run on an enormous variety of systems, and a scientist will think nothing of using four different systems in a single day. She will not even think about the name of the manufacturer (and may have no idea or interest if it is one of a herd of clones of a popular make) and does not want to know anything at all about systems used by her friends when they send mail or files. This insouciance represents on one hand a triumph of portability, but presents an enormous challenge. Systems still provide a huge variety of facilities: multi-tasking, inter-process communication, real-time control, asynchronous input and output, interrupt or exception detection and recovery. The alternatives seem either to program assuming a basic 1965 computer and to ignore twenty

years of progress, or to find some way to encompass this variety. The latter we have so far failed to do.

8. *Patterns of Culture*

Further demands are caused by the world-wide marketplace. It is now common that the hardware must be able to run on 50/60Hz, 110/220V in frigid or blistering temperatures, and must be able to produce a variety of scripts. The demands of international and trans-cultural use are even tougher on the software.

It is not always sufficient for a computer to read and write and speak English. At the simplest level, different sets of translation tables and comment strings and digital sound recordings are needed. At a deeper level, there are cultural assumptions. One was mentioned in the section on time representations. For another, what should the output of an accounts payable system be? In America, it is likely to be a check. In Europe, a wire transfer to a GIRO account or a direct deposit into a bank account is more common. There are complex systems whose only function is to arrange complex barter arrangements for countries with inconvertible currencies. Certain information that is required in databases in some countries is forbidden in others. That fact will surely affect choice of sort keys!

The requirements of transculturalization are really those of program portability writ large: one needs to anticipate the breadth of changes and be prepared to adapt broadly.

9. *Atlas Shrugged*

A very general problem, covering many special aspects, involves the context in which a program runs. Must the program carry its entire world with it? If a program needs to solve linear equations, should it assume the presence of a some particular commercial library, should the package come with a license for the relevant library, or should it contain a copy of the routines needed to do the calculation? How does the program find out about certain system information? Even on similar systems, the locations of

certain files may vary – what directory holds which commands or tables? What names are bad? (It is probably a serious misfortune to have login name "tmp" on a UNIX system.) Some programs can use different methods depending on what resources (temporary files, real primary memory) are currently available, but there is no convenient and portable way to get the necessary information.

10. Pilgrim's Progress

Users are starting to see a glimmering of help.

- Certain popular programs run on a variety of systems, and their users are quite happy without knowing how much work the vendor had to do. The major spreadsheet programs and many computer games are in this category.
- The porting of WordStar to essentially all CP/M machines and PC lookalikes was a triumph of portability and clever work. If a program is sufficiently valuable, massive efforts can be justified.
- The major mathematics libraries have been rewritten with the vector machines in mind, so it is likely that decent if not brilliant performance will be achieved on parallel machines.
- There are (at least two, alas) graphics standards, and there is growing use of a printing standard. Basing programs on them may permit intelligent use of output devices of varying quality without distorting the application program.
- Network standards have been published and are having strong impacts. There are too many of them, and they are still too vague in places (especially at the higher levels of abstraction), but on many systems it is feasible to plug in hardware that masters most of the low-level complexities.
- Some systems are self-identifying: it is possible for a program to discover directly on what hardware configuration or system version it is currently running, so that it can automatically reconfigure itself. Without this information,

programs must resort to extremely unreliable coding tricks for guessing their immediate environment.

At a higher level, there seem to be some points in system design with promise as narrow places to manipulate for standardization. Ideas about user interface are not settled, but some groups seem to reaching agreement on user interface management systems so it may be possible to introduce descriptions of the windowing world and menu-like inputs that can be comprehended by a wide variety of user programs and of hardware implementations. A language like PostScript may provide a low-level printing interface between disparate environments, now that low-cost high-quality laser printers are available that support the language.

11. Look Homeward, Angel

It may be useful to look at some other industry's attempts at standardization and portability for hints about good approaches and objectives. For a moment, consider cameras. These days, most film packages are self-describing, so a fancy camera can tell what speed and color properties it has. Cheaper cameras still require the user to check the box and turn a dial on the camera. However, no camera manufacturer expects you to use its own brand of film (unless the camera is disposable); new formats are introduced with greatest trepidation. The operating characteristics of most cameras are very similar; once you have learned to focus and set apertures and speed settings, most other hobbyist equipment will provide a subset of the features you know about. Some manufacturers force you to use their accessories (lenses, etc.) to maintain quality or profit margins, but many others follow informal standards, so that a wide variety of lenses can be bought and plugged into many cameras. It is necessary to know the family to which your camera belongs, but you can then often ignore the details.

Sound familiar? Many computer users know they have an IBM PC or a clone thereof, but frequently don't remember what kind they bought. They are similarly unlikely to remember the vintage of their Macintosh or other multi-version product. They certainly don't want to think about what release of the underlying system they are running, nor do they buy updates just because

they are offered. It is the responsibility (both technical and economic) of a system provider to handle these problems and not bother the poor owner with them. The objective of portability research and application is to make work as generally available as makes sense at low costs in time, energy, and resources.

The most successful single approach is to define an ideal that contains the substance of the desired product. One should then use tools (homemade or commercial, as necessary) to create specific implementations. The limits of portability are then defined by the scope of one's model and the power of one's tools and the motivation to proceed. The success of the effort depends crucially on the artfulness of the choice of what to include in the model and what to exclude.

References

Peter J. Brown, ed., *Software Portability: an advanced course*, Cambridge University Press, 1977.

Wayne Cowell, ed., Portability of Numerical Software, *Lecture Notes in Computer Science* **57**, Springer-Verlag, 1977.

Ingemar Dahlstrand, *Software Portability and Standards*, John Wiley, NY, 1984.

John Henderson, *Software Portability*, Gower, London, 1988.

Olivier Lecarme and Mireille Pellissier Gart, *Software Portability*, McGraw-Hill, NY, 1986.

Peter J. L. Wallis, *Portable Programming*, MacMillan Ltd., London, 1982.

[*submitted Sept. 4, 1989; revised Dec. 1, 1989; accepted Dec. 18, 1989*]