Implication and Equivalence I/O

Arun Lakhotia

Madras Computer Laboratories
5 B/1 Sixth Cross Street
CIT Colony, Mylapore
Madras 600 004
INDIA

ABSTRACT

We here introduce the idea of 'implication' and 'equivalence' of I/O devices (Sec. I). 'Implication' and 'equivalence' are relations defined between devices. If a device-A implies device-B then operations performed on A are also performed on B. If device-A and device-B are equivalent the operations performed on A are performed on B also, and vice-versa. These device relations are analogous to algebraic 'implies' and 'equivalence' relations (Sec. II). Thus they promise the power of algebraic manipulations on the 'logical' devices. We cite some existing examples of cases where such relations may be defined, though in a primitive form. We conclude by citing some areas of utility for this power.

BACKGROUND

Redirection of I/O is a very well established concept. The term "Redirection of I/O" caught on since UNIX [1] and now is a feature looked forward to in any forthcoming operating system. UNIX provides a facility to redirect an input or output operation requested on the standard input (stdin), standard output (stdout) and/or standard error (stderr) devices to any physical device (other than the default one).

The stdin, stdout and stderr are logical devices which may be associated to any (legal) physical device when invoking a program. The process of redirection is transparent to the program and hence it is possible (and easy) to develop fairly flexible device independant programs.

I. INTRODUCING 'IMPLICATION' & 'EQUIVALENCE' OF I/O DEVICES

Of late we have been feeling the need of some more powerful facilities for providing device independance to programs. We here 'informally' introduce the two ideas (yet)-- Implication and

Equivalence of I/O devices. This article intends to initiate a discussion on the usability, feasibility and viability of such a feature.

Output Implication

If A and B are two output devices and if output(A) implies output(B) then any output operation requested for on device(A) is also performed on device(B).

Thus

* **output(terminal) => output(Printer)**
  anything transmitted to the terminal is sent to the printer also.(effect of ^P sequence in CP/M [2], MS-DOS [3]).

* **output(stdout) => output(diskfile)**
  characters displayed on the screen are sent to a disk-file also.

* **output(stderr) => output(list_device)**
  A copy of error messages is sent to the listing device also.

Output Equivalence

If A and B are two output devices and if output(A) is equivalent to output(B) then if an output operation is requested on device(A) a similar operation is performed on device(B) also. Further any output request made on device(B) is echoed on device(A) also.

* **output(Terminal) <=> output(Printer)**
  would mean that a message that is displayed on the terminal is also sent to the printer. Conversely a message printed on the printer is displayed on the terminal as well.

Note: In order to avoid a non-terminating loop of output (due to equivalence) we need to differentiate between a direct output request and the one caused due to equivalence.

Input Implication

We may define 'input implication' by a definition symmetric to 'output implication' by substituting 'input' for 'output' in the latter's definition.

If A and B are input devices and if input(A) implies input(B) then any input operation that is requested for on device(A) is also performed on device(B).

* **Input(operator's console) => input(terminal B)**
  A program requesting input from the operator's console also accepts input from terminal-B.

Such a definition will obviously pose questions about the program behaviour on receiving two inputs when actually requesting for one. A program would (normally) process only one input request at a time. Further the "implication" process being transparent (and external) to a program we do not expect the program to take care of multiple responses to a single request.

This situation has arisen due to our implicit assumption that the same piece of code is accepting both the inputs. We have walked into it by trying to achieve a functionally symmetric definition to output implication.

Let us redefine it keeping in mind a multi-tasking approach. If A and B are two input device and input(A) implies input(B) then for any task-A requesting input from device(A) another task-B is executed (in parallel) which is a copy of the task-A except that the input requests for device(A) in task-B are redirected to device(B).

* **input(operator's console) => input(terminal B)**
  Terminal B is defined as a parallel operator's console. The commands accepted from the operator's console are accepted from terminal B also. This makes sense when certain previlege commands can be invoked from some specific terminals also.

There being two (different) programs accessing one of device(A) or device(B), the problem of multiple inputs to a single request is solved.

The solution may not be as simple as it sounds because we are yet to study the implication of such duplicity in context with the inherited environment.

Input Equivalence

If A and B are two input devices and if input(A) is equivalent to input(B) then if an input operation is requested for on device(A) a similar operation is performed on device(B) also. Further if an input operation is requested on device(B) it is performed on device(A) as well.

* **Input(res. terminal A) <=> input(res. terminal B)**
  The two reservation terminals A and B are to be treated identically. The requests from reservation terminal A as well as that from reservation

terminal B undergo identical processing.

In spirit of input redirection we'd say that if input(A) is equivalent to input(B) then if task-A requests input on device(A), another task-B is automatically executed (in parallel) where task-B is a copy of task-A except that all input requests on device(A) in task-B are redirected to device(B) also. At the same time if task-A requests an input on device(B) then another task-C is automatically executed, where task-C is a copy of task-A except that all input requests on device(B) in task-C are redirected to device(A).

Such an interpretation has far reaching consequences. Most important being that more than one task requests input from the same device at the same time. Task-A and task-B accept the input from device(B) and similarly task-A and task-B accept the input from device(A). One doesn't have to elaborate the consequences of such a happening.

Let's give another thought to what we 'want' from 'device equivalence'.

We have said that -

> If two devices are equivalent then a program operating on one device performs the same actions on any device that is equivalent to it.

Or can we also say that

> If two devices are input equivalent then the data generated by the two undergo similar operations.

If we accept the last statement we can resolve the '(logic) deadlock' created above by re-stating the equivalence effect as

If two input devices A and B are input equivalent then all input requests made by a program on device(A) and (or) device(B) are treated as input requests on device(A). Further a copy of the task is invoked (parallely) in which all the requests on device(A) and (or) device(B) are directed to device(B).

Such a definition would be a lot more meaningful after we have studied its effect on the inherited environment.

II. DEVICE IMPLICATION AND EQUIVALENCE AS RELATIONS

The real power of device implication and equivalence arises from our recognising them as relations defined over a set of I/O devices (for a particular input or output operation).

Without loss of generality we may associate the relations,

implication and equivalence, with the input and output operation rather than a device. We would then have relations input-implied and input-equivalent defined over the set of input devices, output-implied and output-equivalent defined over the set of output devices.

Our previous notation

output(A) => output(B)

is now equivalent to

A => B

and

output(A) <=> output(B)

is equivalent to

A <=> B.

Similarly we have '=>' and '<=>' for symbolizing input implication and input equivalence.

Here after we would not differentiate between an input or output relation and would just call them as device-implied ('=>') and device-equivalent ('<=>') wherever our statement holds good for either operations.

Device implication is transitive

If A => B and B => C

then A => C

From our definition of input and output implication we can see that if device-A implies device-B and device-B implies device-C then an operation performed on device-A is performed on device-B and an operation performed on device-B is performed on device-C as well. Thereby an operation performed on device-A is performed on device-C also.

Device implication is reflexive

A => A

holds good as an output coming on device-A is sent to itself.

Device equivalence is transitive

If A <=> B and B <=> C

then A <=> C

Any operation performed on device-B is performed on device-A as well (and vice-versa). Further as any operation performed on device-B is performed on device-C also (and vice versa) we can say that any operation that is performed on device-A is performed on device-C also (and viceversa).

Device equivalence is reflexive

A <=> A

is obvious

Device equivalence is symmetric

If A <=> B

then B <=> A.

is obvious from the very definition of device equivalence.

Device equivalence may be achieved by device implication

If A => B and B => A

then A <=> B.

An operation performed on device-A is performed on device-B (due to implication) and an operation performed on device-B is performed on device-A (due to implication). Hence device-A is equivalent to device-B (definition of equivalence).

III. CONCLUSION

The transitive nature of device implication promises a tremendous potential to a device independant program.

Consider the problems like

1) Broadcasting a message to several terminals (or users)
2) Mail/ memo transfer to several network nodes.
3) Handling a number of reservation terminals.

all of them boil down to invoking simple programs with suitable device implications external to it. With the system supporting the work of duplicating output or input the user is left only with the work intrinsic to the problem.

A specific task (though one may consider it too trivial) which may be eased due to device implication is 'error-handling'. Error handling normally requires echoing an error to a standard output device and a standard list device also. With the prevelant techniques this is achieved by writing separate output statements for every device on which the error is to be transmitted. Device implication reduces this to simply one 'generic' output statement, leaving the task of multiplicating the output to the system.

One doesn't have to stress the utility of CP/M ^P facility, UNIX 'tee' and dBASE ALTERNATE [4] facility. These are basically device implication in a restricated domain.

NEWDOS-80 'ROUTE' command [5] is by far the nearest example of defining device implication and equivalence . Using the 'ROUTE' one can acheive redirection, implication as well as equivalence. The input implication and equivalence is not truly what we have defined but a more practical implementation. It resolves the input implication problem by assigning a sequential order on the implied devices. When an input is requested it tests (in order) which device is ready for input. The first device which is found ready is issued the request.

Equivalence may be achieved in NEWDOS-80 by circular ROUTEing.

REFERENCES

[1] Richie & Thompson, "The UNIX Time- Sharing System", CACM 17,7 July 1974, pp 365-375.

[2] "The CP/M Reference Manual", Digital Research Inc..

[3] "MS-DOS", Microsoft Inc., 1983.

[4] Ratliff, W., "dBASE II Assembly Language Relational Database Management System", RSP Inc., pp 80-81.

[5] "NEWDOS-80", Apparat Inc., pp 442-443.

TRADEMARKS