Io: a new programming notation

Raphael L. Levien
Levien Instrument Co.
Box 31
McDowell, VA 24458

10 Sep 1989

Introduction

Many years before the first computer was built, or the first programming notation designed, Alan Turing proved that all computers, and by extension, all programming notations, are capable of solving all the same types of problems (known as computable problems).

What this means in practice is that we can have almost any programming notation we want. Programming notations have been designed to support existing, familiar notations, to be appropriate for a specific class of applications, to resemble English, and countless other purposes.

The author set out to design the simplest practical programming notation possible. The result is the new programming notation Io, which is described in this paper.

Historical background

If the complexity of a programming notation is measured by the number of mechanisms, then clearly the simplest programming notation would consist of one single mechanism.

A typical pitfall of such efforts is to design a notation that is mathematically self-contained, but with no facilities for input or output. In order to use such a notation in practice, it is necessary to forcibly graft input and output commands onto the notation.

In order to avoid this, it is clear that the single mechanism of the programming notation should be able to support communication with the outside world. For this reason, Hoare's Communicating Sequential Processes were considered. However, Hoare's mechanisms are rather complex and awkward, so this was ultimately rejected.

It is also required, of course, that the mechanism also be able to support ordinary computations. In addition, it would be nice if it were also possible to construct large systems out of smaller components, with a mechanism similar to a procedure call.

All of these requirements were reconciled with a mechanism referred to in this paper as 'performing an action.'

Output is accomplished by simply performing an action corresponding to an output device. Input is accomplished by specifying an action for the input device to perform whenever it has an input value.

Buffered and blocked communication can be accomplished by adding a buffer or blocker in series with an action.

Procedure call can be accomplished by specifying a 'return action' to be performed when the procedure is to return.

It is a very simple mechanism, but can be assembled into patterns of any size and complexity. It is interesting that many of the important concepts in computing science show up as particularly simple patterns.

Actions in Io

The main mechanism in Io is the action, which consists of an operator and a number of arguments. An example:

```
write* 5
```

Here, **write*** is the operator, and **5** is the only argument. **Write*** displays its argument on the screen. Thus, when the above action is performed, the screen will show **5**.

This type of mechanism is not particularly unusual in programming notations. What is unusual about Io is that there is no other mechanism.

The Io notation does not include sequence (do A, then do B). This is a problem, because sequence is one of the most important mechanisms in programming.

The solution to this problem is to include sequence in the operator. As an example, the operator **write** takes two arguments. The first argument is a number, which it writes, and the second argument is an action, which **write** performs after it has written the number.

We need a notation for specifying an action as an argument. In Io, this is done differently if the action is the last argument. If the last argument is an action, it is written as

```
; action
```

If an argument other than the last is an action, it is written as

```
(action)
```

So, if we want the screen to show **5 6**, we can perform

```
write 5; write* 6
```

The first argument to **write** is **5**. The second is **; write* 6**.

It is clear that **write*** is not very useful, as it can only be used if it is the last action to be performed. Instead, we can use the **term** (terminate) operator, which takes no arguments, and does nothing. So, our example can now be written

```
write 5;
write 6;
term
```

Taking arguments

We wish to be able to write an action that takes arguments. When the action is performed with arguments, these arguments are bound to the specified variables. The Io notation for this is

```
→ variables; action
```

An action that writes its argument twice is

```
→ x; write x; write x; term
```

We can then define an operator as a particular action. The notation for this is

```
operator: action.
```

So, we can define the operator **writetwice**

```
writetwice: → x;
   write x;
   write x;
   term.
```

Then, when we perform writetwice 12, the screen will show 12 12.

This operator, though, has the same problem as write*; there is no way to perform another action after writetwice is finished. We can fix this problem in the same way: introduce a second argument, here called ret, for return action.

```
writetwice: → x ret;
   write x;
   write x;
   ret.
```

Now, if we want the screen to show 7 7 9, we can perform

```
writetwice 7;
write 9;
term
```

This argument mechanism is also an excellent way for an operator to return values. For example, the operator + takes three arguments. The first two are numbers, and the third is an action. + adds its first two arguments, then performs the third with the result as an argument. Thus, the screen will show 5 after performing

```
+ 2 3 → x;
write x;
term
```

Conditionals

A conditional operator is one that takes more than one action argument, and chooses one to perform. A typical example is the = operator, which takes four arguments. The first two are numbers; the last two are actions. If the two numbers are equal, = performs its third argument. If they are not equal, = performs its fourth.

```
= x y (true-action); false-action
```

is equivalent to the familiar

```
if x = y then true-action else false-action endif
```

A good example of = also introduces an operator that can perform itself recursively.

```
count: → start end ret;
   write start;
   = start end (ret);
      + start 1 → start;
      count start end ret.
```

If we want the screen to show 1 2 3 4 5, we can then perform

```
count 1 5;
term
```

The operators > and < are similar to =. These are illustrated in the operator gcd, which performs its third argument with the greatest common denominator of its first two.

```
gcd: → x y ret;
   > x y (- x y → x;
          gcd x y ret);
   < x y (- y x → y;
          gcd x y ret);
     ret x.
```

Using actions to store data

Here is an action that, in effect, stores the two numbers 242 and 338.

```
twonums: → ret;
   ret 242 338.
```

When twonums is performed, it in turn performs its argument, and gives it the two numbers as arguments.

We can then define writepair, which takes two arguments. The first is an action, similar to twonums, that returns two numbers, and the second is the return action to be performed when writepair is finished.

```
writepair: → pair ret;
   pair → x y;
   write x;
   write y;
   ret.
```

As would be expected, 242 338 is written on the screen by

```
writepair twonums;
term
```

We can also define makepair, which takes two numbers, and return a pair similar to twonums.

```
makepair: → x y ret;
   ret → ret;
   ret x y.
```

When makepair is performed with 242, 338, and some action, it will perform the action with an argument of → ret; ret 242 338, which is identical to twonums.

Although this looks like sequence, it isn't, because the pair returned by makepair can be performed any number of times.

Data structures

By combining a conditional with an action that stores data, you get a conditional that stores data. This can be used to build a data structure.

Take as an example a linked list of numbers. There are two kinds of lists: empty lists, and non-empty lists. A non-empty list contains the first number of the list, and another list, which is the rest of the list.

In Io, a linked list is best represented as an action that takes two arguments. An empty list just performs its first argument. A non-empty list performs its second argument, and gives it the first number in the list, and the rest of the list.

We can define writelist, which takes two arguments: a list and a return action. It writes all numbers in the list, then performs its second (return) argument. As described above, the parentheses notation is used for action arguments other than the last argument.

```
writelist: → list ret;
   list (ret)
    → first rest;
   write first;
   writelist rest;
   ret.
```

Here are a few operators for constructing linked lists. Note the similarity between makepair, above, and append-to-head, below.

```
empty: → null full;
   null.
append-to-head: → num list ret;
   ret → null full;
   full num list.
append: → list0 list1 ret;
   list0 (ret list1)
    → first0 rest0;
   append rest0 list1 → rest0;
   ret → null full;
   full first0 rest0.
append-to-tail: → list num ret;
   append-to-head num empty → list1;
   append list list1 ret.
```

Performing

```
append-to-tail empty 1 → list;
append-to-tail list 2 → list;
append-to-tail list 3 → list;
writelist list;
term
```

will show 1 2 3 on the screen.

Binary trees

Because Io has no assignment statement, it would seem very difficult to insert a value into a binary tree. However, the solution is quite simple: create a new binary tree containing the new node.

A binary tree has similar structure to a linked list. It is performed with two arguments. An empty tree, like an empty linked list, just performs its first argument.

```
emptytree: → null full;
   null.
```

A non-empty tree will perform its second argument with the key, the data value, the left subtree, and the right subtree.

The operator search performs its third argument if the key is not in the tree, or its fourth argument, with the data value, if it is in the tree.

```
search: → tree key notf found;
   tree (notf)
    → k d left right;
   < key k (search left key notf found);
   > key k (search right key notf found);
      found data.
```

The operator ins (insert) takes a tree, a key, a new data value, and a return action. It returns a new tree, with the new data value inserted.

```
ins: → tree key data ret;
   tree (ret → null full;
         full key data emptytree emptytree)
    → k d left right;
   < key k (ins left key data → left;
            ret → null full;
            full k d left right);
   > key k (ins right key data → right;
            ret → null full;
            full k d left right);
      ret → null full;
      full k data left right.
```

Coroutines

Coroutines are an important concept of computing science, but few programming notations properly support them. It is surprising how easy they are to implement in Io.

The idea of coroutines is to have two (or more) routines. When one of the routines gets to a point where it can no longer proceed (such as, when it needs more input), it is suspended, and another routine continues until it, in turn, can no longer continue (such as, when it has a value to output). Then, it is suspended and another routine is resumed.

This is used, for example, in creating a stream. A stream carries a sequence of numbers, without consuming storage. Therefore, it can be infinite. Even in the case of a finite stream, though, it has an advantage over a linked list, because computation can begin immediately after the first number is known.

The Io implementation of streams is analogous to linked lists. A stream takes two arguments. If there is no more data in the stream, it performs its first argument. Otherwise, it performs the second argument, with a data value and the continuation of the stream.

Here we define the operator count-stream, and bind an infinite counting stream to the variable s.

```
count-stream0: → x out;
   out x → null out;
   + x 1 → x;
   count-stream0 x out.
count-stream: → ret;
   ret → null full;
   count-stream0 0 full.
count-stream → s
```

S has exactly the same structure as a linked list. In fact, writelist s will write 0 1 2 3 4 5... on the screen.

Concurrency

Concurrency is achieved with the par operator, which takes two action arguments. It performs both actions simultaneously.

Since there is no assignment to variables, there is no chance of conflict from improper use of shared variables. On the other hand, there is no possibility of communication between the two actions. We need a facility for this.

Concurrent communication

As mentioned earlier, the mechanism of performing an action is similar to concurrent communication. Performing an action corresponds to sending a message, and an action that gets performed, in a sense, receiving a message.

The difference between synchronous communication, as Hoare describes in his CSP, and Io, is that if a process in CSP is not ready to receive, the sender must wait until the receiver becomes ready. In Io, the receiver is always ready. If more than one message is sent at the same time, then, in effect, multiple copies of the receiver are created.

In the case where the receiver corresponds to a physical device that can only handle one request at a time (such as a disk drive), some facility is necessary to arbitrate, and only let through one message at a time.

This is accomplished in Io with the chan built-in operator. Chan takes one argument, to which it immediately returns two actions, known as the send and receive actions.

When either the send action or receive action (but not both) is performed, nothing happens until the opposite is also performed. But, when this happens, the argument to the send action is returned to the argument of the receive action. Only one such pair is matched at a time.

This gives us enough to implement write, given an operator screenwrite, which takes a number to write, a number for the cursor position, and an action to which it returns the new cursor position. In order to allow several different processes to write numbers on the screen, so that they don't collide, we can define

```
chan → write! write?;
writemon: → cursor;
   write? → x;
   screenwrite x cursor → cursor;
   writemon cursor.
par (writemon 0);
write: → x ret;
   par (write! x);
   ret.
```

In cases where more data than a single number is to be transmitted, an action containing data can be used as the argument to the send action.

A note on implementation

An important consideration in any programming notation is how close the mechanisms are to the implementation platform. This is mistakenly known as the speed of the notation.

For example, C was originally designed partly as a fancy assembler for the PDP-11. Mechanisms such as *p++ correspond exactly to PDP-11 instructions.

Performing an action in Io is actually similar to a goto. However, passing an action as an argument is similar to closure formation. In a naive implementation, a dynamic allocation would be necessary for every time an action was used as an argument. However, even this is not too expensive, and garbage collection is not required.

Because Io is such a simple notation, the prospects for a very efficient implementation are good, even without a complex compiler.

It seems likely that a large proportion of the time spent by a naive implementation would be for the dynamic allocation. This can be greatly reduced by combining the space required for several (or many) closures into one dynamic allocation.

It may also be possible to allocate some space in a stack, if the platform does this efficiently.

Another important optimization is to code some operators in-line, rather than to code a goto or call to the operator. This is particularly important for the built-in operators.

With these and other optimizations, it should be possible to implement Io at least as efficiently as other programming notations.

Another intriguing possibility is the compilation of Io programs directly to hardware. The idea of performing an action is probably best implemented with transition signalling. As a matter of fact, the author got the idea for Io immediately after reading Ivan Sutherland'sTuring Award Lecture about transition signalling and micropipelines.

Conclusion

Io is a very simple programming notation. It turns out to be surprisingly powerful as well. Without using any built-in operators, it is possible to construct boolean values, linked lists of arbitrary type, numbers (implemented as a record of boolean values), and binary trees of arbitrary type. In addition, still without any built-in operators, it is possible to traverse linked lists and trees, as well as implement coroutines. This is not a bad track record for a programming notation with only one mechanism.

With the addition of built-in operators for efficient implementation of numbers, input and output, and concurrency and communication, Io becomes a practical general purpose notation.

C.A.R. Hoare said, 'simplicity is the price one must pay for reliability.' Perhaps, with Io, simplicity is not such a high price after all.

References

[1] C.A.R. Hoare, *Communicating Sequential Processes,* Prentice-Hall International, London, 1985

[2] Ivan E. Sutherland, *Micropipelines,* Turing Award Lecture, Communications of the ACM, 32,6 (June 1989), 720-738

[3] Guy L. Steele, *Lambda: The Ultimate Declarative,* AI Memo 379, MIT AI Lab, Cambridge, Nov 1976