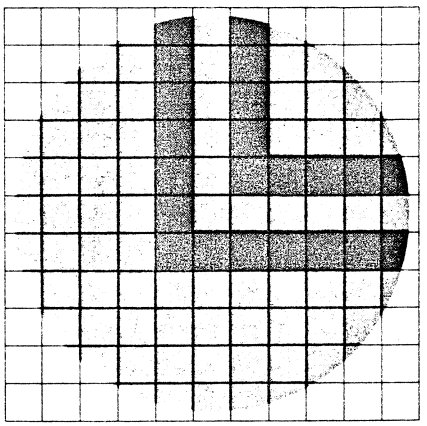
Lattice Logic
6 Albany Lane
Edinburgh EH1 3QP
Scotland
(Registered Office)

Telephone
031-557 3215



Dear Hamish,

We would like to thank you for spending the time and effort in producing IMP CONVERGENCE II, and feel that your new views can only help to resolve this particular problem which has been with us for far too long.

While we broadly agree with many of your proposals there are particular concerns about some topics and we would like to put our arguments into the general discussion. In addition there are several points which it might be a good idea to discuss in the context of producing a more complete definition of the language.

The enclosed notes roughly follow the order of your memo.

We look forward to an early discussion on the future of IMP.

Peter S. Robertson

Ian Young

Peter S. Robertson
Ian A. Young

cc G.E. Thomas, P.D. Stephens, G.E. Millard (ERCC)
P.D. Schofield, S. Michaelson, D.J. Rees (Computer Science)

Objectives

An important point to consider from the outset is efficency. There are two quite separate areas:

1. The inherent efficiency of language features.
2. The efficiency of the implementation of language features on particular hardware.

Arguments about efficiency are dangerous at the best of times, but those based on specific hardware capabilities must be treated with extreme caution, unless it can be shown that any subsequent changes to the language would be likely to give an improvement on the majority of processors.

From the point of view of the general efficiency of the language it must as far as possible retain the property that if you don't use a feature you don't pay for it.

There is a tendency in many languages to provide magic forms which look like normal procedure calls but which cannot be defined within the language. Obvious examples are the I/O procedures in Pascal: READ(X); READ(F, X); READ(X, Y) etc. Currently the only example in IMP is the map ARRAY. We feel that it would be a retrograde step to introduce more of these 'procedures'.

The IMP compiler for the Interdata 7/32 (developed by Chris Whitfield) showed that with a little thought many run-time checks can safely be omitted. In particular it was quite common for that compiler to remove over 60% of potential unassigned variable checks. This can improve the performance of programs without increasing the chances of undetected errors generating obscure bugs.

Whatever the final design, there is a pressing need for an extensive test/validation suite of programs to aid compiler development and to provide some measure of conformance evaluation.

Proposals

(a) All 'magic' procedures should be removed or the language should be extended to make the means of their definition available to users.

(b) Existing features and proposed additions/changes should be carefully examined to make sure that they do not compromise the ability to optimise diagnostic checks.

(c) A test suite should be designed.

The IMP language

We wholeheartedly support the idea that a proper manual should be produced. Publication by EUP would be a welcome bonus and might provide the incentive to take a more careful approach than perhaps has been taken with purely internal documents. As Lattice Logic is actively involved in the dissemination of the language we would be happy to help in the production of the manual.

Predicates

The only real solution to this problem would be to include booleans into the language, however, this could not be done lightly as it would have major difficulties, for example what would this mean:

```
boolean B
integer X, Y
....
if X # 0 then B = X < Y and Y = 3
```

Perhaps a group could investigate the possibility of introducing these very odd objects.

We are opposed to permitting a single arithmetic expression as a simple condition. PSR implemented this several years ago and it proved to be a prime source of obscure bugs - 'some loss of error-detecting redundancy' was in reality a great loss. A typical example of the sort of error introduced was typing:

```
if X - 1 start     instead of    if X = 1 start
```

with exactly the opposite effect (many terminals have '=' and '-' on the same key so it is very easy to get the wrong one).

Proposals

(a) Predicates should be introduced
(b) Expressions as simple conditions should not be introduced.
(c) A working party should consider booleans.

Integer ranges

It is very difficult to define ranges in a flexible and consistent way; just look at the mess Pascal got into. We believe that properly-defined ranges are invaluable and must be in IMP. Notwithstanding the problems (such as those which lead to LENGTHENI etc.) we agree that ranges should be implemented immediately, following the style of (and inheriting the problems of) bytes, shorts etc. Some of the benefits we see are:

1. the possibility of saying what you mean.
2. the opportunity for more checking, both in terms of the range itself and in some cases use of otherwise illegal values for unassigned checking.
3. the opportunity for code optimisation.

We are not happy with the proliferation of keywords: byte, half, long, mite, nibble ..... and would prefer to be able to give a name to a range:

```
range Sbyte(-128:127)
integer(Sbyte) X, Y, Z
```

Also it would be convenient to be able to access the bounds on the ranges supported by a particular compiler, possibly by the introduction of pre-defined constants: (say) Plus Infinity, Minus Infinity. Another approach would be to permit the ubiquitous star as in:

```
integer(0:*)  Positive
integer(*:-1) Negative
```

Real precision & ranges

Some thought is needed here to prevent similar machine-dependent meanings of real, longreal, longlongreal etc.

Records

Partial record assignment can be achieved in VAX IMP in the following possibly cumbersome but safe way:

```
recordformat Smallfm(integer A, B, C)
recordformat Bigfm( (record(Smallfm) Small or integer A, B, C),
                    integer X, Y, Z)
record(Smallfm) Small
record(Bigfm)   Big

Small = Big_Small
```

The alternative format is to allow use of Big_A rather than Big_Small_A. The only problem is that there is no guarantee that the user has made Big_A == Big_Small_A etc. However there is no possibility of corrupting something unexpected when the partial copy is made. Our objection to the indiscriminate "move the smaller number of bits" is that it is totally uncontrolled.

Explicit component specification in records

We object to this on the grounds that it can be an expensive feature to implement yet the cases where it would be useful are not common and the benefits are trivial.

Comparison of records

This seems a good idea but the definition has problems. There seem to be two obvious possibilities:

1. Comparison on a bit-by-bit basis (c.f. record assignment).
   This will give unexpected inequality in the following case:

```
recordformat F(string(15) S)
record(F) A, B
A_S = "1234567"
B_S = "abcd"
A_S = B_S
if A = B start
```

   The problem is the debris left from the first assignment to A. The only way round this particular problem is to clear out the trailing junk on every string assignment - surely a waste of time in the vast majority of cases.
2. Comparison on an element-by-element basis, using = or == (# or ##) as appropriate. This gives results as distressing as the previous definition when alternative formats are about.

Proposal

As some form of record comparison would be useful what about definition 2 above with the limitation that records with alternative formats cannot be compared. In addition, compilers may optimise the comparison into a bit-by-bit one when strings are not present.

Strings

VAX IMP accepts the form: conststring(*) Day = "MONDAY"
We fail to see why the star is inappropriate in this context, as the actual length is made explicit in exactly the same way as when a string pointer variable is pointed at a string of known maximum size. Indeed it would even seem reasonable to allow:

```
string(*) Data = "123456"
```

where the maximum length would be set to 6.
There is no need for three forms of string declaration: string(12), string(*) and string; the same applies to records. The (*) form was added to make the difference from Emas IMP explicit. It would be possible to remove the (*) form altogether and simply use string instead, but we favour the use of star to "say what you mean".

LENGTH seems to be used as a map in two contexts:

1. To truncate strings. This could be done using the TRUNCATE function mentioned under 'jam-transfer'.
2. To get the effect of S=S.Tostring(Sym), usually in a misguided attempt to produce 'better' code.

We see three problems with LENGTH being a map:

1. It limits the scope for different implementations of strings, in particular there would be no possibility of having strings longer than 255 characters.
2. Being a map, LENGTH must take a string(*)name parameter. This removes the possibility of testing the parameter for being unassigned unless the compiler makes use of compile-time information not available for user-defined maps (i.e. the fact that the reference to LENGTH actually generates a value). For example:

```
begin
   string(15) Text
   integer Count = Length(Text)
   Write(Count, 1);  Newline
endofprogram
```

   This program outputs the value 128 (or some similar magic number) without giving an unassigned fault.
3. LENGTH can compromise the optimising of string capacity checks. For example:

```
string(15) X, Y
.....
X = Y
```

   The assignment should not need to be checked as the maximum length of Y is defined as being no greater than that of X. However if the check is omitted a previously executed LENGTH(Y)=255 could cause havoc.

Proposals

(a) LENGTH should become an integer function.
(b) CHARNO(S, Index) should remain a map but with the restriction that 1 <= Index <= LENGTH(S). This implied call of LENGTH will provide the unassigned check and will ensure that CHARNO cannot corrupt other data.

Scope rules

There are currently problems with code sequences such as the following:

```
integer X
begin
   integerarray A(1:X)
   integer X
   .....
```

We understand that Emas IMP is inconsistent about this, faulting the example given but accepting the equivalent where X is a procedure (a dreadful fiddle beloved of manual writers). The obvious step is to consider that use of a global identifier is effectively a local definition of that identifier and fault any subsequent attempt at redefinition. At the very least a warning should be produced by the compiler. This point should be given careful consideration especially as Pascal encourages (forces) forward references in this manner. The only objection we can see to this is that some method is needed for passing 'intrinsic' procedures as parameters. The solution is to do it properly in the compiler, not to require users to write obscure code sequences.

Loops

The VAX IMP form of for statement was also defined with efficiency in mind.

1. (trivially) it results in a comparison for equality which on several machines can be compiled much more efficiently than a comparison for greater or less.
2. (much more important) if the final value of a for statement control variable is undefined it seems sensible to make it unassigned to prevent compiler-dependent programs being written. This has the unfortunate property that variables can become unassigned and hence checks which would otherwise have been omitted must be included. For example:

```
integer J = 0, K, L
                {J need never be checked for unassigned}
.....
J = K           {K must be checked here}
A routine call
L = K           {but need not be checked here}
```

It has been stated that the for statement is simply a packaged form of a more cumbersome loop, in which case it is inconsistent to state that the control variable is undefined at the end.

The arguments against general use of repeat until instead of a combination of exit and repeat seem to be of the "we think it's bad for you" type, similar to arguments which led to no exit, result or return statements in Pascal.

Typographical and syntactic variations

1. Line-breaks in strings are convenient - it's a pain to have to contort text in the cases where line-breaks occur naturally.
2. We think a hyphen for continuation is a great improvement.
3. It has been pointed out that unless braces nest as comment delimiters they cannot be used to comment-out sections of code:

   e.g. `if X = 1 {first case} or X = 10 {last case} start`

4. map is now an unfortunate term as most maps are not store mapping functions but truly name functions.

Operators and literal expressions

IMP has quite enough operators as it is, and the existing ones are reasonably obvious (with one or two exceptions such as ! and !!). We think it would be a bad thing to add more operators which would either be obscure (:=:, ><, ...) or would be keywords which look distinctly odd. The clear solution is to make certain functions known to the compiler. This is both open-ended and readable, although PSR can forsee certain problems in implementation.

Dollar as a prefix for the TOSTRING operation has been implemented and we dislike it. Dollar is too insignificant on the printed page for the function it is performing, and the meaning is not obvious. In addition, dollar is the obvious character to act as a 'double underline' to control a macro processing front-end which would give the effect of conditional compilation.

Name arrays

1. It is not just Computer Science users who favour them; they are widely used.
2. There are many other features in IMP which are not capable of consistent extension; the discussions about namename parameters have been going on for years.
3. When PSR was dubious about implementing name arrays he tried the "the same effect can be achieved" argument and received the reasonable reply "why can't I say what I mean".
4. We strongly object to changing the meaning of objects 'on the fly' in some circumstances. For example, what is the meaning of the assignment in:

```
recordformat F(integername P)
record(F)array Pt(1:10)
        {instead of integernamearray Pt(1:10)}
Pt(1) = 0

Does it mean: Pt(1)_P = 0
         or: Pt(1)_P == Integer(0)    ?
```

Proposal

name arrays should be implemented

Pointer variables

Surely an external spec is the wrong way of defining the location of a variable. Why not simply use:

```
integer Clock at 4
external record(DCBfm) DCB at 16_800
```

We agree that the untyped name variable is problematic, but we think that there is a need for it in several contexts (notably in implementing heap storage). The only real need for the TYPE OF function is in the routines READ and READSYMBOL. Perhaps the solution here is to replace:

```
      READ(name N)

with  READ INTEGER(integer(*)name N)
 and  READ REAL(real(*)name N)
 and  READ STRING(string(*)name S)
```

READSYMBOL could then take an integer(*)name parameter, banning a real parameter - not a common usage! Sadly this would bring back the badly-named READ ITEM family of procedures.

Overloading procedure identifiers has at least one major difficulty. Some encoded form of the parameter specification must be included in the external references generated when overloaded externals are used. This would be almost impossible on systems with primitive external linkage facilities (most IBM systems = modified OS360), and on many others would lead to incomprehensible errors messages when an unknown parameter combination was specified.

Arrays

IMP77 carefully did not define the mapping from array declaration to the actual layout of storage so as to be able to take advantage of varying machine characteristics. Multi-dimensional arrays can be seen as arrays of arrays (as in Pascal) and this leads to a natural order of evaluation of the subscripts and the possibility of reducing the dimension of an array, for example when passing it as a parameter. Although we do not like the idea of initialising *own* and *const* multi-dimensional arrays with unstructured lists of values (because the correspondence between subscripts and values is esoteric), if no better solution can be found the most obvious ordering must be used.

Array names and maps

We agree that the whole business of arrays and array mapping must be cleaned up, but we are not sure exactly what was offered as a suggestion. The following is our (modified) reading of it:

An array is to be considered as a two component object:

1. a linear sequence of consecutive objects of the same type

2. a mapping from a number of integer index values to those objects.

An *array* will both create the objects and define a mapping. An *arrayname* will only define the mapping; the objects will be located at the time the *arrayname* is associated with an *array*. With this approach the provision of a function returning the number of objects in the array would facilitate the writing of fairly general procedures. (SIZE OF would seem the obvious candidate but the result would need to be divided by the SIZE OF a single element). In order to simplify the definition of *arraynames* a star could be used in place of one of the bounds, the actual bound being deduced at the time the *arrayname* is pointed at an array. Once such a star has been replaced by the appropriate value can the *arrayname* be pointed at an *array* of different size, or is the definition subsequently fixed?
The built-in map ARRAY now has two parameters: the number of elements in the array and the address of the first of them. This still has the minor irritation that ARRAY is an *arraymap*, an object that users cannot define.

Compiler control

Control is an archaic feature which should be removed (or left for compiler debugging and development). *Option* is much cleaner.

There are two ways to go with conditional compilation. One is to embed control structures into the language, and the other is to provide a more powerful macro pre-processor. In either case a dollar would be a convenient way of introducing the control words ($IF, $ELSE etc).

Events

The event mechanism is proving to be of great use and is being used much more often than was envisaged when it was defined. As a consequence the mechanism must be as efficient as possible. The problem of the range of event numbers is more serious than the discussions to date have suggested. Very few programs that we have seen trap any of the language-defined events except input ended. Most users cram several meanings into the small number of free event numbers, and this leads to obscurity and confusion. Invariably users of packages of procedures have to include complex code sequences to trap the one event they want while letting others past. We would like to see the range of events increased dramatically. For example, make all language-defined events fall in the range 0:14, allocate events >=15 to packages on request and leave negative events for general use. We can predict that the objection to this will be that it will make events less efficient (there are conveniently 16 bits in halfwords!), however an examination of the existing IMP77 compilers has shown that this aint necessarily so.
To obtain the maximum benefit from a large number of user-defined events it would be necessary to keep a central register of ranges of events allocated to specific programs: Edwin, ILAP etc.

The case for "on event *" is that there are several layered systems in which all events must be trapped (for example a BASIC interpreter, or even an Emas-style subsystem). Even in simpler programs it can be convenient to catch anything going wrong and tidy up before stopping. If the previous suggestion of extending event numbers is accepted then complete enumeration becomes impossible.

Permanent procedures

It is crucial for a standard library of procedures to be defined as rigorously as possible. This is an area where variations in operating environments have made themselves most noticable and unwelcome. We feel that the arbitrary description of procedures as 'permanent', 'external', 'semi-permeable' etc. is not as important as the detailed specification of their properties, parameters, results etc. Parameter specifications are most important especially as most if not all existing IMP compilers/linkers/loaders do not check that the parameters on the spec match those in the definition. A good example here is ItoS. Whether or not a spec is necessary is insignificant when compared to the effects resulting from using ItoS(N, Places) on Emas and ItoS(N) on VAX. From an efficiency point of view the list of pre-declared objects should be kept as small as possible to reduce the overheads in compiling every program. Many of the variations in this list from system to system have come about because certain procedures were frequently used and so were included for general user convenience. Perhaps such procedures could be incorporated automatically (in a system-dependent manner) from a user-supplied include file with a (system) standard name.

String Resolution

It has been suggested that string resolution be redefined yet again as follows:

> If a destination is omitted the string fragment which should have been assigned to it must be null or the resolution fails.
>
> If a destination is to be ignored it must be specified as *.

Heap storage

For some time now versions of IMP have supported heap storage in a simple but effective way using the two procedures:

```
record(*)map NEW(name N)
routine DISPOSE(record(*)name N)
```

with references such as:

```
recordformat Fm(integer A, B, C, record(Fm)name Link)
constrecord(Fm)name Fm Type == 0

record(Fm)name P

P      == New(Fm Type)
P_Link == New(Fm Type)
....
Dispose(P)
```

While these have proved invaluable they have several problems:

1. Fm Type is needed because recordformats are very much second-class objects; it can only be used to declare records and cannot be used in SIZE OF or passed as a parameter. In general it would be nice to be able to get the size of a format without having to declare a dummy variable. One suggestion has been to permit the format identifier as a name with an unassigned address.
2. There is no way of ensuring that the parameter to NEW, which is used solely to find the size of area required, bears any relation to the ultimate use of the area. This could be solved by the use of the long-awaited namename parameter: New(P). This would also remove the necessity for the dummy Fm Type variable and the restriction that NEW and DISPOSE only manipulate records.