WG11/N325

Problems of Language Bindings

Brian Meek

[The views expressed here are those of the author and do not necessarily reflect a UK view, a UK bindings panel view, or a WG11 view. The paper is intended as an individual contribution to the continuing debate on cross-language issues.]

Introduction

A *language binding* is the definition of way in which a facility or concept defined in a language-independent way is realised in a particular programming language. The concept first came to prominence when the standard for the Graphical Kernel System, GKS, was being developed, and bindings were needed so that GKS functionality could be invoked by programs written in a variety of languages. Since then, bindings of other kinds have been developed, or are in the course of development.

A binding can be defined in a separate standard, or incorporated in one of the standards directly. Normally one would expect either a separate standard or a addendum to the language standard to be produced, later perhaps to be incorporated in a revision of the language standard, either within it or as a separately published "Part 2" or whatever. It has been suggested that a language-independent standard could incorporate language bindings in, say, a normative annex, though as yet there are no instances of that approach.

It is of course logically possible for a product to conform both to a language standard and to a language-independent standard, defining its own binding. However, conformity to a binding standard is preferable, since otherwise bindings for different products could behave differently in some circumstances, e.g. where there is interaction with optional or implementation-dependent features. Of course, in the absence of a binding standard, implementors have no alternative but to define their own (or, perhaps, produce their own, since they might not define it explicitly). This demonstrates the importance of developing the language binding standards along with the language-independent standard, or as rapidly as possible thereafter. It is the language community concerned that will suffer, either through problems caused by diverse bindings or by being deprived of the benefits of the language-independent standard. Implementors, or at least the professionally responsible ones, are more likely to use the language-independent standard if the binding is defined for them, but could be more reluctant to do so if they have to work out the binding from scratch. (Of course, they might already have or wish to produce a quite different proprietary solution, but that is a hazard of all standards, not just binding standards, and is no more a problem here than everywhere.)

A language binding must of course conform both to the language standard and to the language-independent standard. The logical possibility therefore arises that the two standards conflict - that it is not possible to conform to both simultaneously. The developers of language-independent standards will normally make every effort to avoid this. The trouble usually is that this gets done in such a way that the standard is undermined, typically by the use of options, or weakening conformity requirements. However, often it can be done, albeit with greater effort, without such undesirable side effects.

Harder to avoid is designing the language-independent standard with unspoken, and quite possibly unconscious, assumptions about the host language, based upon the languages the standardisers themselves happen to be familiar with. This shows the importance of the developers of the language-independent standardisers consciously trying to recruit people with different language backgrounds, and for the language standardisers to take an active role in the development of the language-independent standard, rather than just criticise it afterwards. (It has to be said that neither of these things has been very evident to date.)

A related difficulty is when the original form of the language-independent standard was an implementation in a particular language. Inevitably the design of the standard will reflect the nature of the language, and design decisions, including perhaps fundamental ones, will have been influenced by the language used. This bias (intended as descriptive and not a pejorative term) can be carried over into the language-independent form and even find its way into bindings to other languages. At least, however, the known background serves to alert people to the possibility of such bias and to allow aspects that might cause difficulty in other languages to be identified.

Different kinds of binding

Functionality new to languages

As stated earlier, binding first came into prominence in relation to adding functionality, defined in a language-independent way, to languages that did not have it. ISO/IEC TR 10186, *Guidelines for language bindings*, arose out of the experiences of binding graphics standards to various languages. The SQL-Ada binding, adding database access facilities to Ada, is a later example, and the language bindings, existing or envisaged, for Posix and PCTE are in the same category. However, three other distinct kinds of binding exist, and it cannot be assumed that yet more will not arise in future.

Features included in languages but left undefined

The next kind to add to the first is where the language-independent standard features exist in languages but its provision is left mostly or wholly implementation-dependent. The Language Independent Arithmetic (LIA) standard is the archetypal example of this kind of standard, which requires its own style of binding. The standard was developed because the language standards failed to specify adequately the requirements on arithmetic processing needed to ensure predictable results within a given range of accuracy. The binding of the LIA standard to a language hence amounts to expressing the LIA requirements in the terms of the particular language standard - in effect, tightening the specification of arithmetic operations and stiffening the conformity requirements.

Thus the binding effectively means little more than using and perhaps extending the terminology of the language standard, and taking into account any requirements on arithmetic (such as minimum ranges to be supported, detection of exceptions such as overflow, etc) that are already present. If additional functionality is needed to meet the LIA requirements, this is an incidental byproduct, and anyway likely to be minor.

In some cases it is possible to add a "soft" definition of the LIA requirements expressed in the language itself (e.g. in the form of procedures), though this is not really necessary except perhaps to aid understanding in the language community of what "added value" the LIA binding will bring, and to aid testing of existing implementations against the LIA requirements.

In theory, with this kind of binding a language which did in fact specify requirements on arithmetic might include some which conflicted with those of the language-independent standard. In the case of LIA this does not seem to be a problem in practice - concerns have been expressed about LIA, but have arisen for other reasons. To date, the only specifically language-related issue concerned with binding arose as "what happens if the language standard specifies some of the functionality but not all?". In LIA this arose in the context of languages which do not include a datatype **real**, i.e. one with approximate numeric values and operations, but the issue is of more general applicability.

A completely general answer is difficult, because it is an instance of interaction between the conformity rules of the language-independent standard and the language standard. If the language standard permits functional extensions provided these do not conflict with the requirements of the standard, as is commonly the case, then it is fairly straightforward. The binding standard would say that the features that the language does not cover would have to

be provided in such a way as to meet the relevant language-independent standard requirements, and that if the missing features are provided in an allowed implementation extension they too have to meet the relevant language-independent standard requirements - pointing out that such an extension has to be provided if conformity to the language-independent standard itself is to be achieved.

Where such extensions are not permitted by the language standard, or are not possible without conflicting with it, the situation is more tricky. You could still specify a binding for the features which were covered, citing the relevant parts of the language-independent standard, but omitting the rest. If the rest consists of missing functionality, then it might be possible to emulate it within the language, but that might not be feasible, practicable or worth the effort.

You would then have a binding which did not conform to the whole of the language-independent standard, but that is not unusual; for example, IT standards commonly do not use only the terminology of the vocabulary standard ISO 2382, but use the definitions they need and add their own.

If the users of such languages are not to be deprived of whatever benefits the language-independent standard will bring to those features that the language does provide, the conformity rules of the language-independent standard need to be framed to cover such situations. On the other hand, they must not be made so lax that people can just choose the bits they like, violate the rest, and still claim conformity - as seems rather the case with terminology. It should not be possible, for example, for a binding to LIA of a language which does have both **integer** and **real** datatypes to claim conformity on the basis of one but not both; that debases the concept of conformity to the level of worthlessness. Such a standard is useless if implementors can pick and choose which bits to use and not bother with the rest, yet still be able to claim conformity. If the language-independent standard does permit partial conformity, it should be on the basis of "the binding for all of the features which the language does cover must be done as required".

Of course, it is still open to a standards committee (perhaps a language committee) to produce a binding standard which does pick and choose in that way, adopting from the language-independent standard the specifications and requirements it needs, and providing its own for the rest. However, it is unlikely that this would occur without good and documented justification; it is better than leaving it to an implementor's whim. In that case, neither the binding standard nor products conforming to it could claim conformity with the language-independent standard, something which the binding standard would need to point out explicitly.

This analysis shows that, one way or another, difficulties caused by incomplete coverage of the language-independent standard features can be coped with, while still avoiding so weakening its conformity requirements that its value is eroded.

Many of the concerns which have been expressed about the LIA standard, and language-independent standards to be mentioned later, have arisen because of people not making it clear precisely what kind of conformity of what kind of product they are concerned about. Concerns have been expressed in some language communities about whether "the language" can conform (whatever that is supposed to mean), rather than the language standard, or a language binding (binding standard), or language products seeking to conform to the language-independent standard as well as the language standard. There has been some confusion inside people's minds, or between minds of people with different unstated assumptions of what conformity of what to what they are talking about.

Not just unstated assumptions, one suspects, but unformulated and even unconscious ones as well.

Functionality defined in both standards

The third kind of binding arises where functionality defined in the language-independent standard is, commonly, mostly or wholly already defined in the language standard as well. The aim here is not "to reach the parts that the others do not", but to achieve commonality, to aid the provision of common language-independent services to and by products, systems and services in different languages. The term "functionality" is used here, rather than the more general "features", since this kind of binding is aimed at achieving the desired commonality in the case of processing specified in program text, and performed by executing the code. This turns out to be a different kind of binding problem to achieving commonality of generic conceptual features, which is the topic of the next section.

The archetypal functionality which all language standards define is input-output, and this would have ptovided the ideal example, if a language-independent standard for I/O actually existed. After all, a moment's thought will show that it is absurd that every language standard should have many pages devoted to specifying the syntax and semantics of I/O. Alphanumeric and binary I/O at least cries out for a language-independent standard which language standards can reference, merely citing (as a binding) the language syntax to obtain the functionality needed by the language.

This situation does not pertain, and there is little prospect that it will for a very long time. Not that alphanumeric I/O varies all that much between languages (binary is a different matter); after all, languages share the same I/O equipment and interfaces to it, and implementors tend to be able to patch the facilities demanded by the language standard into those provided by the hardware and operating system platforms. However, the language standards have their own I/O specifications because they were developed in different isolated environments, and to an extent too large to be desirable still serve ingrown communities unaware of, indifferent to, or even distrustful of, the world outside. Hence there is little incentive to develop a language-independent standard for I/O, however much sense it makes in the abstract.

Nevertheless it is still worth considering how a language binding for an I/O language-independent standard would be defined. In many cases, perhaps most, it would simply be a matter of specifying what language-independent standard functions are invoked by what language commands. If the language-independent standard conformity requirements were similar to those of the second kind of binding - everything you cover that is in the language-independent standard you must do that way, but the rest can be omitted - that might be enough. However, the binding standardisers, perhaps to meet some demand in the language community, might feel it worthwhile to cover as well the language-independent standard functions not already in the language standard; even without a specific demand, it might be justified, to avoid implementors doing it anyway but in different ways.

In that case means of binding could be a module, for a language extensible by such means; a set of procedures (which some languages use for I/O anyway); or a language extension specified as an addendum.

The possibility still remains that a language might have I/O which does not lend itself readily to conversion into language-independent standard form, or requirements that are incompatible with the language-independent standard. In that case, a solution might be to provide both the language-independent standard and the native language I/O facilities in parallel. A precedent for this, on a smaller scale, has existed for many years in Fortran, where the Fortran 77 standard maintained the previous stance that the character set (provided it included at least a specified repertoire as a minimum), and its collating sequence, were implementation-dependent, but added functionality to provide the collating sequence implied by ASCII coding as well.

It should not be necessary to go to such an extreme for languages to bind to the principal example to date of a language-independent standard for language-defined functionality, that being developed for procedure calling. This standard is aimed at enabling a procedure call to be "answered" in a different language: i.e. the formal parameters and procedure body are in one language but this defined procedure can be invoked from programs written in other

languages. Such facilities are already provided on some implementation platforms, but in a system-dependent way, so neither the procedure libraries (which is what is usually involved) nor the programs that call them are portable.

The concept is that the actual parameters get "marshalled" by the binding to the client language into a language-independent form for passing to the language-independent standard procedure calling service, which unmarshals them into the form required by the formal parameters, using the binding to the server language. The definition in the language-independent standard of what a procedure call is, and the various forms of parameter passing explicitly or implicitly assumed or allowed by languages, are believed to be sufficiently comprehensive to make specifying the binding a straightforward matter. Essentially, the marshaller and the unmarshaller for a language, specified in the binding, are import/export interfaces to a conceptual common language-independent environment which allows the marshaller of one language and the unmarshaller of another to communicate.

The reason why bindings of functionality defined in language standards can be done in a relatively straightforward way is that the processes which programs cause to take place when the functionality is used exists at the operational level of abstraction: running a program causes things to happen. One way of looking at it is that all language processors ultimately are Turing machines, and what any one can cause to be done, so (with greater or lesser difficulty) can any other. The situation is rather different at the conceptual level of the language's realm of discourse, i.e. the linguistic level of abstraction (see [Meek 1992] for a more detailed discussion of different levels of abstraction). That is what we shall look at now.

Concepts defined in both standards

The archetypal example of a concept defined in an language-independent standard and in language standards is that of a datatype. In many ways this can also be regarded as the most fundamental of all, since all languages are about the processing of data. (This applies even in so-called "untyped" languages, where the lack of typing refers only to absence of association between language entities and particular datatypes. Datatyping is there even if the only datatype a language recognises is an eight-bit byte.)

The processes of calling a procedure, passing parameters, and returning results can be discussed in terms of the way they work, as in the last section, without reference to the values of the actual parameters that are passed across, i.e. without reference to their datatypes. Indeed, in some "weakly typed" languages the actual parameters may be passed willy nilly (i.e. without type checking) and the recipient procedure may take these as found, and convert them to its needs, or reject them as unusable. It is well known that in such languages the onus is on the programmer writing the procedure call to ensure that the actual parameters are of the kind required.

(This is not intended as a criticism; it is also well-known that the freedom from type checking in such languages can be exploited to achieve some effects very simply which would be very much more complicated to achieve in a strongly typed language - even though it may need skill to get them as intended, and care if unexpected effects are to be avoided when a normal call is all that is wanted.)

The effect of having a language-independent standard for procedure calling without reference to the datatypes of the parameters would be to impose weak typing in this part of the common environment. Worse than that: in a weakly typed language environment a programmer can at least ensure that actual and formal datatypes match, but here it might not even be known what language the formal parameters come from. There are ways round that; but worst of all, leaving parameter datatyping issues as implementation-dependent would effectively reduce to zero the benefits that language-independent standard procedure calling could bring.

That alone would be justification enough for a language-independent standard for datatypes, but datatyping is so fundamental a concept, as already mentioned, that it needs separate

definition, and one not confined to the needs of parameter passing. Data in databases have datatypes and language bindings to database standards involve them. They arise when the data output by a program in one language is to be read by a program written in another language. (Indeed it can be seen that any language-independent standard for I/O would meet the same issue).

However, binding the language to the language-independent standard for datatypes is a different matter from binding to functionality like I/O or procedure calling, because of differences between the linguistic and the operational levels of abstraction. It affects the very fabric of the language, not just the processing actions a program causes to take place. It is easy to imagine alternative functions for "native I/O" and "language-independent standard I/O", and even alternative forms of procedure calling (they have actually been suggested); but having alternative definitions for "native" and "language-independent standard" datatypes really makes little sense - even if the language communities would accept it. Either there would be effectively two languages sharing the same control structures and syntax but with different datatypes (apart from any overlaps where the definitions happened to be the same, e.g. **Integer** and **Boolean**), or there would need to be conversions available. Conversions involve another binding, which immediately shows that the "alternative form" kind of binding is not enough.

Indeed, for functionality like I/O and procedure calling, where it may seem to be enough, such an "alternative form" binding will still have to depend on the existence of a datatype binding, based upon conversion, if it is to be effective.

Hence one is driven to the conclusion that the form of binding for concepts like datatypes built into the language at the linguistic level, which are defined both in the language standard and in the language-independent standard, can only be a mapping, the definition of an interface between the language and the common environment. For full generality, there needs to be defined mappings for both directions. In some cases you would need only one: for writing a file, for example, or reading one written by an unknown source.

Since either mapping may be one-many or many-one, in reciprocal situations (such as an in-out procedure parameter), the entity passed through the interfaces may need to "remember" its origins so that the appropriate return path can be chosen. But those aspects can be left to the functionality and services using the mapping - they exist at the operational level and can be specified as requirements in the appropriate definitions. At the linguistic level, static inward and outward mappings suffice. Specifications at the operational level can select the aspects they need, and their standards determine the conformity rules.

It can be seen that, in cases like this, it is not "the language" or the language standard which conforms to the language-independent standard - unless someone were to define a language using the language-independent standard specifications directly. (It is certainly possible to imagine a language using for its datatypes a selection of those in the datatypes language-independent standard.) In general, what will conform to the language-independent standard (and to the language standard too, of course) are the mappings, the definitions of the interfaces. To avoid conflicting alternative mappings, the binding standard hence must specify the mappings and provide the conformity requirements. Products providing language-independent standard services in such cases would not conform directly to the language-independent standard, but indirectly through conforming to the binding.

Hence direct conformity to the language-independent standard would be mainly confined to the various binding standards, and any other standards (typically at the operational level) depending on it. This is exactly the case, with respect to datatypes, for the language-independent procedure calling standard, which will conform to the datatypes language-independent standard in respect of procedure parameters. However, direct conformity of products of any language-independent standard must not be ruled out; quite the reverse, since the hope is that in future unnecessary diversity will be avoided by direct implementation of the language-independent standards in new products. Just because at present all products

are likely to conform indirect;ly is no reason to exclude definition of direct conformity from a language-independent standard.

Who should do the binding?

If those are the different kinds of binding that are needed in different circumstances, who should be responsible for producing the binding standards, and does the answer vary between the different kinds?

In trying to answer these questions, one crucial aspect needs to be taken into account, namely the different targeting of the two standards to which the binding must conform, the language standard and the language-independent standard.

Language standards have traditionally been aimed at programmers, to specify what they should write to produce a standard-conforming program. Apart from the requirement to process standard-conforming programs in accordance with the standard - i.e. to use the semantics specified (and sometimes specified rather vaguely and leaving a good deal of latitude) - the standards have not primarily been aimed at implementors. There has been a shift in recent years and with newer language standards towards including more requirements on implementors, but the emphasis and underlying culture in the language standards community has remained on the programmer side.

Language-independent standards, on the other hand, are aimed entirely at implementors. They may embody requirements *of* users but they consist of requirements *on* implementors, to build language processors, interfaces or utilities that meet the language-independent standard specifications.

It is clear, therefore, that bindings of an language-independent standard to languages must also be aimed primarily at implementors. The programmer enters into consideration if a program needs to be written in a way that will invoke the language-independent standard features. This is likely to be the case if the language-independent standard specifies functionality for which provision in the language did not previously exist, but may not apply otherwise, for example calling a procedure in a library using an language-independent standard interface may be no different from calling one in a native language library; in the colloquial phrase, adding this facility would be "transparent to the user".

A further factor to be borne in mind is that, for a binding to be acceptable, it must be appropriate to the language. If user (programmer) invocation is explicit, the form of invocation must sit naturally in the language; and the implementation of the binding must fit the way the language is implemented. The first, though not trivial, is relatively straightforward to achieve, but the second may not be.

It is often said that bindings should be done by those who developed the language-independent standard, since they understand its requirements. That is certainly true as far as it goes, but it means that the language bindings are likely to be variable in quality, depending on the level of expertise available in the languages concerned. Where a functionality standard is concerned, a particular language may already have been used for its implementation, and it would be foolish to ignore that experience. For functionality where a procedure library or a library module is a likely form of binding, the language-independent standard experts could well develop a generic binding, a template to be picked up and used, by whoever produces a particular language binding standard. A generic binding could even be included in any language-independent standard, if those responsible felt able to provide it. But to ensure appropriateness it is hard to see how involvement of the language community - in effect, the language standard committee - can be dispensed with.

The more the language-independent standard covers features already embedded in the fabric of the language, the more true this is. The argument that the language-independent standard group should do the bindings is strongest where new functionality is involved, i.e. the first kind

of binding discussed at the outset and where this thesis first developed. It is almost as strong for the second kind, though more language community involvement is necessary since a binding defining what has previously been left undefined may have side-effects which need to be evaluated, something which only that community can do. If (as it is quite likely) a given side-effect may be seen as beneficial by some and the reverse by others, only the language community can resolve that issue. It may depend on why the feature was left undefined; it may have been thought unnecessary; it may have been thought too difficult to define; it may have been a deliberate omission to resolve conflicts, e.g. between different implementations.

In the last case, however, these would have been implementations at the time the standard was developed. These implementations may now have been superseded, the implementors (or supplier organisations) originally objecting to a particular definition may no longer be in the marketplace. In other words, the original reasons for leaving something undefined may no longer be valid; the language-independent standard binding provides an opportunity for the issue to be revisited. In the first two cases, the language-independent standard provides a ready-made definition which can be used "as is" without any re-inventing of wheels.

The argument that only the language community (committee) can decide applies even more strongly in the third or fourth cases. In the first case, and perhaps also the second, the language binding could be developed by the language-independent standard group and merely reviewed for approval by the language groups - the practice to date. Even in the first case the result has not been invariably happy. It is hard to see how it could work at all in the third and fourth cases; the result is more likely to be that the work is wasted, or has to be done twice. For, as we have seen, the language-independent standard and its bindings are aimed at implementors, and implementation experience among the language-independent standard experts for a particular language is likely to be the exception rather than the rule. What language experience there may be is more likely to be programmer experience, and probably not at the expert programmer (in that language) either. Such programmer experience is valuable, of course, but not enough.

In addition, the burden of producing bindings on behalf of a number of language communities is likely to be very great, and will all come at the same time - as the language-independent standard is being finalised - if maximum benefit is to be obtained. Indeed, the various language bindings, at least where an immediate use for the language-independent standard can be identified, ought ideally to be developed along with the language-independent standard. They may lag a little behind in development, that is probably inevitable, but should appear alongside or shortly after the language-independent standard itself. This approach also means that any difficulties encountered in doing the binding can be fed back into the language-independent standard work and lead to improvements.

All of these factors point in the same direction - that the bindings should be done in the language community, though perhaps with the aid of a generic binding provided by the language-independent standard experts. They can then ensure that the binding is appropriate to the language and provides what the language community needs. No-one else can do that for them.

Furthermore, it is clear that the right people to do it are the implementors in the community. The language-independent standard is aimed at implementors, and if language products are to incorporate language-independent standard features the implementors will need to understand the language-independent standard and the binding standard as well as the language standard. Users need to be involved as well, of course, but primarily to ensure that their requirements are met.

But *will* they do the binding?

If it is the implementors among the language community who should take the lead in developing the binding standard - and, one way or another, get involved in the language-independent standard work itself - the question remains whether they *will*. While

implementors - and the suppliers who employ them - may have an interest in implementing language-independent bindings in some cases, factors may intervene which will inhibit them participating in the binding standard work:

1. They may want to carve out a market niche for a particular language-independent facility in a particular language or languages, and not wish to let others share it. (But standards are about sharing.)

2. They may be willing to spend time (even their own time rather then their employers') on currently fashionable and "glamorous" languages but not be motivated to put so much effort into established, "bread and butter" languages. (The need there may be as great or greater, and the task harder because of the accumulated weight of history, but with no sense of compensating interest and excitement in tackling a new area.) Hence there is no guarantee that there will be people coming forward for all of the standardised languages for which bindings are needed.

3. There is traditionally a general resistance by implementors to have standard specifications of things like this; they prefer to be left free to do it their own way. (They tend to be very eloquent in advancing the advantages to users in this approach, while brushing aside the disadvantages of incompatibilities and inconsistencies that are likely to arise.)

On balance the likelihood is that only a relatively few implementors in a language community would be prepared to get involved. They may want an language-independent specification, even a standard - but would want any bindings left implementation-dependent. This of course is not good enough to provide users with the predictability of behaviour between platforms and between languages, that they have a right to expect (even if too many have become accustomed to, or even brainwashed into, not expecting it, or even questioning its absence).

This reluctance is predicted not just because of the evidence of experience to date, but because of the long-standing cultural tradition of compiler-writing, only gradually being eroded by the passage of time and the slow percolation of "open systems attitudes" into hitherto largely closed and introverted language communities. It is compounded and shored up by the fact that this isolationist attitude, the "one language mentality", can be at its worst among some users, rather than among implementors. The culture and tradition inside language communities is that the language is complete unto itself, beholden to nothing outside, with wheels invented and hand-crafted afresh for each one. Hence the indifference to, suspicion of, even outright hostility to language-independent standard activities that can be found even within language standards committees, never mind the wider language community outside.

Yet in the long run language-independent standards can make the job of the language committees easier, by providing language components ready made, to be used as appropriate, instead of having to go through a long and perhaps painful design process. In the past the response to suggestions that they should do bindings, or even review those done by others, has often been "we've got far too much to do, we can't take that on as well". Just as a large software engineering project can be made easier by using procedure and module libraries, so a language standards project can be made easier by use of language-independent standards - or at least existing components where no actual standards yet exist. The one is regarded as good professional practice; the other as yet seems to be anathema.

Conclusion

If things go on as they are, language bindings will not be agreed, or at least be a long time in coming, and when they eventually come may well be flawed because the right people have not been doing the work. The losers will be the language community, whether they are aware of it or not. If things are not to go on as they are, something must happen to cause change, and for the reasons given earlier it has to come from within the language community itself.

Despite what has been said above, the place to look is the language standards committee. The reason for this is that this is where there are people who at least are aware of standards and their importance and benefits (apart from the occasional participant who is there not to help develop the standard but protect a vested interest).

It is therefore up to these committees and working groups, and the users on them, to take up the cause of language-independent standards and persuade implementors within their community to take on the binding standards, for the benefit of the language community as a whole. The problems are not trivial, but effective solutions are possible, provided they come from within the language communities concerned. Within those language standards groups is the place where this process must begin.

References

ISO/IEC TR 10186 Guidelines for language bindings, forthcoming

[Meek 1992] MEEK, B.L., Programming languages: towards greater commonality, DECUS symposium, Solihull UK, May 1992

WG11/N330

ANNEX TO WG11/N325

Programming Languages: Towards Greater Commonality

Brian L. Meek
Computing Centre
King's College London

Let us begin with a few observations, some of them self-evident, all of them well-known and readily verifiable.

> There are a large number of different programming languages, in serious active use by significant numbers of people, most of them available on a variety of hardware and operating system platforms.
>
> Most of these differ markedly one from another in style, substance and appearance, so much so that usually the language concerned can be identified from only a small fragment of code.
>
> These differences can be accounted for by the fact that numerous different criteria can be specified as potentially desirable, some of them mutually conflicting, and very different languages tend to result from the choice of criteria and the relative priorities assigned to them.
>
> Nevertheless all languages have basically the same purpose, to allow programmers to express what they want an IT system to do, and even very disparate languages have much more in common than is obvious on the surface.

This paper seeks to argue that too much emphasis is placed on the differences, while commonality has never been properly exploited. An analogy, whose aptness may surprise some people, is the way differences in origin, culture, environment etc between people, too often cause their underlying commonality as human beings to be ignored. The aptness is not total, of course; if people were like programming languages, then to relieve headaches, "aspirins" with different chemical formulae - not just different words for the instructions on the bottle - would be needed for every community on earth.

Providing language-independent facilities to relieve headaches caused by unnecessary diversity in languages is a difficult business. The diversity may be unnecessary, but that does not mean it is easy to cope with, let alone remove. Historically, new languages have arisen where people, sometimes just individuals, have perceived that no existing and available language met the needs of their application or environment and that it was easier to develop their own. It does not matter whether or not they were right in thinking it was easier, nor even in thinking already available languages were unsuitable. It does not matter if their false perception was based on ignorance. It does not even matter if all they were doing was rationalising a desire to create their own thing even though this was demonstrably unnecessary. They did it, they redid all the design (usually from the raw hardware up), made many fundamental design decisions based on their intended application or facilities, or the hardware available, or just *ad hoc*, and built them so deep into the fabric of the language that they became impossible to change.

In many cases - most - these artefacts wither away and die, or else survive on a small scale in isolated corners where conditions are not too hostile, like so many weird and lovely (or unlovely) plants. These need not concern us. We are concerned with the still sizeable minority which gain a following, and start spreading in use and influence.

That some do is of course accounted for by the third general observation that we started with. But usually there comes a time when use of the language is no longer confined to its community without interaction with the outside world, in particular the world of other languages. Implementors encounter problems when the new language's view of the world differs somewhat from those of the host platform or other languages already supported. Users have problems when they try to exploit the facilities. Programmers and their employers have problems when they have to adjust to or retrain for the unfamiliar language. Often the minor differences cause more trouble than the major and hence obvious ones; things are taken for granted which ought not to be, and you only find out later, often the hard way, when things don't work as expected.

Implementors have traditionally tended to deal with this by bending the language to suit the platform, or how they thought it ought to have been designed rather than how it was, hence leading to dialects and further problems for users, even those using just the one language.

It is the job of standardisation - proper, official standardisation that is - to provide means of coping with or alleviating the effects of incompatibilities. Again the traditional approach is to do it wholly within the language community concerned, i.e. dealing with the dialect problem but not the inter-language problem. However, a few years ago the ISO/IEC subcommittee responsible for programming languages, JTC1/SC22, set up a working group, WG10, to look at issues of commonality. As well as producing guidelines for programming language standardisation, which I as WG convenor had the task but also the privilege to edit, this group also looked at areas suitable for cross-language standardisation, where underlying commonality could not just be exploited but bring significant benefits.

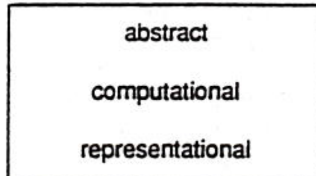
There were several contenders, of which more will be said at the end, but two stood out as being highest priority: datatypes and procedure calling. All languages specify the processing of data and all, even those sometimes called "untyped", have some recognition of different types of data, e.g. that numeric data is different from textual data; while all have somewhere the concept of specifying a "procedure" (not always called that) and invoking it from elsewhere in the program. When eventually these topics were approved as standards projects, they were assigned to another SC22 working group, WG11, entitled "language bindings" for historical reasons not relevant here. Much of this paper will be about these projects and related work.

Before embarking on discussion of these projects it is worth saying a little more about general issues. Language independent standards (or specifications) may be *base standards*, which specify the basic but non-linguistic building blocks from which languages are made; *functional standards*, which add functionality which the languages otherwise do not have; and *generic standards*, which provide a specification in a language-independent manner of concepts which all (or many) languages have in common. Note that these terms are used here not in the rather specialist senses in which they are applied to Open Systems standards, though there are similarities arising from the everyday meanings of the words *base*, *functional* etc. OSI does not have a monopoly of these terms, and functional standards were being talked of in connection with languages long before OSI's profiles were ever dreamed of!

In the current context, character sets are the archetypal base standards, whereas graphics standards like GKS are the archetypal functional standards. Datatypes and procedure calling are clearly in the generic category. Note that this classification is rather of roles than categories; a particular standard may have more than one of these roles, or assume different roles for different languages (e.g. a standard may be generic for languages with certain facilities built in but functional for those that do not).

A second important distinction is between *levels of abstraction*. Language definitions and hence standards, being machine-independent, should exist at a level higher than that of representation on actual machines. On the other hand they cannot be purely abstract, because ability to represent and manipulate is a prerequisite. Mathematics, at the abstract

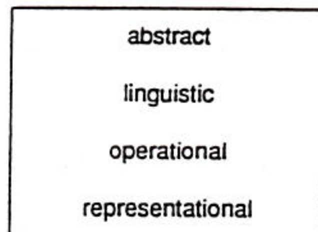
level, can readily deal with infinite sets, for example. So between the two there is this "computational" level:



Language-independent datatypes and procedure calling both definitely belong to the computational level, though unfortunately some actual languages confuse the issue by have explicit or implicit representational assumptions, and/or specify things in too abstract a way, which means that incompatibilities can arise.

An example might be the definition of **Integer** as the (infinite) mathematical domain at the abstract level. At the computational level it becomes a finite subrange *lowerbound:upperbound* of **integer**, with constraints, for example on the magnitudes of the bounds and the need to handle both positive and negative values (too many language definitions ignore this). In an actual implementation it will become an actual subrange with (if you are lucky or the standard is good enough) known actual bounds so you know what you are getting.

Even this last is logically above the level of actual representation, though often driven by what is convenient to represent, and it turns out that it is often useful to make a further distinction between two sublevels of the computational level:



By happy chance the pyramidal image produced by the lengths of these English words neatly illustrates the extra detail you need as you drop down the levels. Troubles arise if levels of abstraction are confused.

Let us now turn to the projects themselves.

Common Language Independent Datatypes (CLID)

This generic standard, currently (in mid 1992) at draft stage as ISO/IEC JTC1 CD 11404, provides a reference collection of datatypes which can be used as common ground by actual programming languages. CLID is an enabling standard, to aid the specification and development of tools and services which all languages can share, and exists at the linguistic level of abstraction. For languages to avail themselves of such CLID-based facilities, mapping standards will be needed to bind the native datatypes of each language to CLID, and vice versa. The CLID standard specifies conformity rules for such mapping standards and provides guidance in performing the association. In particular CLID specifies mappings of type to type, and of value-of-a-given-type to value-of-a-given-type, but no more. The CLID standard specifies "characterising operations" for values of each of its types but it is not a

requirement that a language must support all of those operations for values of the native datatype associated with the relevant CLID standard datatype; that would be dropping to the operational level.

It may then be asked, why does CLID mention operations at all? The answer is simply to help those attempting to define mappings between actual languages and the CLID standard datatypes to recognise the most appropriate match for their purpose - the "best fit". In general, for each datatype the characterising operations listed are neither exhaustive nor minimal. The aim has been to provide enough characterising operations to distinguish a given CLID standard datatype from others with a similar "value space", and leave users of the standard (envisaged after all as typical and experienced "language people") in no doubt as to the nature of the type intended.

It must be borne in mind throughout that the CLID standard provides a very general conceptual model not aimed at any specific application. Despite some misapprehensions, it does not exist just to support parameter passing in common language-independent procedure calling, the other project referred to and described later, though that is undoubtedly an important application. Similarly it does not exist just to support transmission of data between one language processor and another (though that too is important).

An implication of the CLID standard is that if language-independent facilities define their own datatypes then this multiplies the number of mappings that have to be defined. If an application needs more than one such facility then this number is multiplied again. Types so defined are likely to be limited to those required for the facility and may be further constrained because the facility requires a representational model to be added. This could lead to a situation where, in some applications or environments, one facility is restricted by the datatypes supported by the other.

Basing all such language-independent facilities on the CLID standard datatypes avoids the need for a multiplicity of mappings and reduces the danger of one facility being constrained by the limitations of another, at least in respect of datatypes. It is of course important to ensure that the CLID standard is sufficiently general to support all kinds of language-independent facilities.

An important characteristic of CLID is that it is not based on any particular language style or paradigm. (That word has to be worked in somewhere these days to ensure that the paper is seen to be intellectually respectable.) The primary design decisions were that it had to be rich in datatypes ("maximalist" rather than "minimalist") and support strong typing. Richness is necessary so as not to reproduce the "representationalist" error of forcing things into a predetermined mould; you may need to do that at the data transmission level, for example, but not here. Strong typing is necessary because it is easy to get weak typing by relaxation of rules and automatic type conversions, but not easy to do the reverse. The strong need not use their strength; the weak have no choice.

The result is that CLID has a wide variety of constructed types, like aggregates of numerous kinds, a rich variety of primitive types from which to build them, facilities for specifying derived types such as subranges, and means of specifying new, logically distinct instances of existing types. We did stop short of including object-oriented datatypes, as being too open-ended, though a future revision might include them. We stuck to types with operational properties with in general no semantic connotations. (An exception to this is **date-and-time**, since some languages have it explicitly; but some of us are uneasy about it!) For the moment, the provision of object-oriented datatypes has to be done indirectly by the facilities in CLID to construct and declare additional types not explicitly defined in the standard.

Even types with only two values are distinguished one from another, even though the transformations between them are simple - **bit**, with values *0* and *1* and operations + and *; **boolean**, with values **true** and **false** and operations **and**, **or** and **not**; **state**, with two suitably named values such as *yes* and *no* or *on* and *off*; **enumerated** of length two, also with

suitably named values; and **range** *0:1* **of integer**, with all the integer operations but (unlike with **bit**) the possibility of overflow. Of these, **boolean** and **state** are unordered, while the others are ordered - another characterising feature.

Further discussion and explanation of the CLID approach, using this illustration, can be found in [Meek 1990].

The approach, and the whole project, has encountered a variety of attitudes, as might be expected. As alluded to earlier, language communities have traditionally tended to be rather parochial and ingrown, and many people within them regard the project with indifference, ranging through suspicion, to outright hostility - whatever their language, it looks alien (which in many ways it is, of course!). Interestingly, the most general positive support for it in the standards world has come from outside the language committees - but then they know all too well the difficulties of dealing with a multiplicity of languages with incompatible views of the world. Gaining acceptance is needing a long, patient process of education and re-education, that it is not a threat and not an overt (or even a covert) attempt to undermine their world and make them add a mass of outlandish, unfamiliar datatypes to their language. But old-established cultural traditions die hard.

Language Independent Arithmetic (LIA)

While CLID datatypes in general are readily defined in terms of their sets of values, there is one common datatype for which this is not straightforward, namely **real**, the approximation at computational level of the mathematical domain of real numbers at the abstract level. The reason is of course that the values themselves are approximate, which you cannot specify precisely at the language level unless the language has means of requiring a fixed-point representation. CLID defined such specific fixed-point subsets of the real domain as **scaled** and so in CLID **real** does mean the non-fixed approximations - usually floating-point, of course, but for CLID exactly how the approximations are arrived at and represented is not relevant. The characterising features of **real** in CLID are that the exact values are not known at the linguistic level; neither are the results of applying the arithmetic operations.

This brings us to the next language-independent standard project in WG11, one not previously mentioned because it began outside the official standards world and only joined WG11 when the project gained official recognition.

The motivation for this project was that as far as **real**, i.e. approximate arithmetic is concerned (at the operational level now, not linguistic), users still cannot rely on hardware design engineers taking fully into account all of the factors that numerical analysts would wish, especially with respect to minimising accumulated errors. Even if the accuracy of approximation of the original values can be controlled, the accuracy of the computations carried out on them cannot. Again it is partly the cultural heritage of the designers and implementors - speed at all costs, even at the expense of accuracy. For many applications, on modern hardware the approximations to values are close enough that, even if the computational accuracy could be improved, the resulting errors are not large enough to matter - for example they may still be swamped by experimental errors in the original data. However, for some applications it *does* matter, and one *does* need the best possible achievable accuracy, or at the very least for the accuracy to be predictable. Experience shows this not to be the case; and of course predictability is specially important if one wants to write software which is portable between different platforms.

In fact it is not, generally, that hardware is incapable of meeting quite strict requirements with regard to accuracy and predictability. The problem for users is usually a combination of hardware design and the way that implementors of languages and packages exploit it. But it is not unreasonable for users to wish that the hardware will not even *permit* the software writers to use it in an arithmetically unsafe way - or at the very least make it much easier for

them to use it safely than unsafely. Yet still there are no standards to require this. The aim of the Language Independent Arithmetic (LIA) project is to provide such a standard.

There is, to be fair, one honourable attempt which went a long way towards this goal - the floating-point standard ANSI/IEEE 754:1985. This does provide the required properties, including means for detecting spurious results. Since its inception it has been widely adopted for the powerful workstations increasingly used in the late 1980s and 1990s by professional scientists and engineers. There are, indeed, some who advocate its universal adoption, as a permanent solution to the unsafe arithmetic problem.

However, the IEEE standard is not just an arithmetic standard but a hardware architecture standard, albeit an abstract one; it may have become popular for workstations (though its penetration into other areas has been limited), but there is no need to specify hardware architecture precisely in order to achieve safe arithmetic. That is another example of confusing levels of abstraction and purposes of standards. Perhaps even more important, while the IEEE standard does indeed make it *likely* that arithmetic will be safe, it does not actually *guarantee* it, because of that perpetual bugbear of IT standards, the presence of options. Sensible use of the options will preserve arithmetic safety, but you cannot be *sure*, and it is above all *certainty* that users need. Even on the same IEEE-conforming hardware, significantly different arithmetic results can and have been obtained for the same computation carried out using different compilers - *for the same language*.

LIA's approach is to define an abstract set of required properties for arithmetic operations and functions which will ensure arithmetic safety. It is therefore at the operational level and hence complements CLID as far as datatype **real** is concerned. It has representational aspects only to the extent that it is expressed in terms a floating point format - you need that amount of "representation" to specify anything at all. However, this does not mean, and LIA does not require, that underlying hardware supporting a language implementation has to provide it, though as things are today people, if not the standard, might expect it. For the purely language point of view, how the requirements are implemented is irrelevant, even if they may be of interest to the programmer for a particular application.

The LIA standard is in three parts, and the first part, on the basic arithmetic operations, is available in draft form as ISO/IEC JTC1 CD 10967; the other two parts, on mathematical procedures and complex arithmetic, are still being prepared. An early draft of part one, under its original title of "Language Compatible Arithmetic", is available in the general literature [Payne, Schaffert and Wichmann 1990]; while many details have changed during the formal development to CD 10967 stage, the basic principles can be obtained from that. Further discussion may be found in [Wichmann 1990].

Because of its implications for hardware and its potential impact on numerical software, LIA part 1 has engendered a good deal of interest and comment, some of it quite trenchant (see for example [Kahan 1992]). Doubts were raised about motivation, in that one of the large suppliers was providing a good deal of support for the project. It is not unusual for substantial involvement in a standards project by one major supplier to be looked at with suspicion by other suppliers and by users! In all branches of politics "guilt by association" is sometimes resorted to as a line of argument so such concerns can perhaps be dismissed, unless of course backed up by technical arguments. There are enough instances in IT of standards being little more than a standards body wrapping round a thinly-disguised product manual, but even the earliest versions of LIA are a far cry from that.

Some critics have argued that LIA would undermine IEEE 754, or weaken it in some way; supporters of LIA argued in return that it actually strengthened IEEE 754, and the combination of the two standards was the best that could reasonably be hoped for given the state of the art in the 1990s; yet it also offered hope and support to those who for one reason or another did not have the benefits of IEEE 754 to support them. In any case, if the IEEE 754 approach is so good, how can this really be a threat? This surely hints of protecting something weak

rather than robust! Or is it simply wishing to be dog in the "safe arithmetic" manger, or taking umbrage at anyone straying off the One True Path?

In fact, very likely most of it is the result of nothing more than simple misunderstanding of the aims of the project. In particular, *there is no such thing as an LIA machine* in same sense as that there is undoubtedly such a thing as "an IEEE machine", and once that essential point has been grasped then some of the objections should be defused. Another red herring has been concern that LIA is so "weak" as to permit approximate represemtations which are inadequate for "serious" numerical analysis. This also misses the point: safe, predictable approximate arithmetic calculations are needed for a wide range of purposes, and not all of these would qualify as "serious numerical analysis". If a representation which numerical analysts would scorn is nevertheless adequate for some purposes, why should users be deprived of the protection that conformity to LIA provides? It is not for the LIA standard to dictate to users what they should or should not use in the context of a given application, only that the arithmetic performed meets certain criteria of accuracy.

Fortunately the debate has also thrown up useful constructive points which will improve the end product. As at mid 1992 it looks hopeful that outstanding concerns can be sufficiently cleared up in time for LIA to progress to DIS (Draft International Standard) stage, meaning that the technical content has been finalised, by the end of the year or early in 1993.

Common Language Independent Procedure Calling (CLIP)

As mentioned earlier, procedure calling was the second of the topics identified as a priority for a language independent standard. As with datatypes, the purpose of the CLIP standard is to provide a common reference point to which all languages can relate. Again, it is an enabling standard to aid the development of language-independent tools and services; again, mappings to actual languages will be needed; and for parameter passing it will need to use the CLID standard, which would have been required for this purpose even if it were not needed otherwise.

It will aid the development of common procedure libraries. Operating systems very often contain such libraries, of things like the mathematical functions, which are shared by the various language processors running under them. However, they tend to be embedded in the environment and to be system-specific - in other words, not portable between platforms. The CLIP standard will enable such libraries to be specified in a standard way and built in a standardised language such as C or Fortran. It can be noted that numerical procedure libraries will be able to make good use of all three of CLIP, CLID and LIA. Portability between platforms is of particular importance for language-independent functional standards like graphics, which can be implemented in this way even though the procedure calling may be disguised when used from inside a particular actual language.

Another use of CLIP is to aid mixed language programming; in fact it was demands for this to be supported, in a standardised way portable between platforms, which led to the work item proposal. In mixed language applications, called procedures would run on language processors operating in "server" mode, and the procedures would be called from language processors operating in "client" mode. Note that the languages need not be different, and if the processors are the same the model collapses into conventional single processor programming. However, one can envisage some applications using a language processor with good diagnostics or a good human-machine interface to call procedures written in the same language running, say, under an optimising compiler or a vector processing server. (Hence the term should really be "mixed language processor computing" though that is rather clumsy.) In such a use of CLIP, the procedure calling mappings would be straightforward but not totally redundant; the CLID-based parameter passing would still usually be needed as a filter for incompatibilities caused by the use of optional features or implementation-dependent aspects allowed by the language standard. This could be dispensed with only if it were certain that the processors shared identical characteristics in every relevant respect.

CLIP of course exists at the operational as well as the linguistic level because it is a dynamic and not just a static concept. The full title is in fact "Common Language Independent Procedure Calling *Mechanism*". In some papers is therefore abbreviated to CLIPCM or CLIP-CM, though others use CLIPC or, as here, CLIP (CLIPC is more accurate but CLIP is shorter). The word "mechanism" is not incorrect, since what it refers to is a *language* mechanism. Such a concept is understood (usually) in programming language circles but open to misinterpretation outside - it tends to lead people into thinking it is an *implementation* mechanism as well.

If that is what CLIP is for, why is a standard needed? There would of course have been no need, had commonality already existed. But whereas most if not all programming languages include the concepts of procedures and their invocation, they vary in the way that they view them, especially with respect to parameter passing - and this irrespective of any differences there may be about datatypes.

In Appendix B (RPC tutorial) of the ECMA-127 standard for remote procedure calling (RPC) using OSI (Open Systems Interconnection), it is argued that "the idea of remote procedure calls is simple" and that procedure calls are "a well-known and well-understood mechanism...within a program running on a single computer". Both statements are true at a given level. Procedure calling is a simple concept, at the level of provision of functionality. It becomes less so at the linguistic level (let alone the operational and representational or implementation levels) because of its interaction with both datatyping and program structure, as is testified by the numerous variations and (apparently) arbitrary restrictions on procedure definitions and calling in existing languages. It is also undoubtedly true that procedure calling is "well-understood" almost universally in the language community. The trouble is that this general understanding is not necessarily the *same* understanding.

This is to some extent acknowledged in section 3.4 of Appendix B of ECMA-127. However, once these points are put together, along with the absence hitherto of CLID, the "simple and well understood" view of procedure calling becomes increasingly elusive; more tenuous, harder to sustain.

The CLIP standard specifies conformity rules for language processors operating in client mode and in server mode. It is expected that many processors will conform in both modes. Taking the CLID datatypes for the parameter types, what it adds are language-independent operational definitions of different kinds of parameter passing, based upon the "contract" concept that the client "undertakes" to supply a particular item of that datatype (e.g. a value or a reference) at a certain time (e.g. when the server "accepts" the contract, or when the server, while executing the procedure, requests it), and the server similarly "undertakes" to return the results of execution, in the form of changes to parameters or (in the case of "function" procedures, as an overall result (though that can trivially be identified on the server side as an extra "out" parameter).

It is important to note that CLIP does not address the question of *how* this contract is reached - how the procedure call initiated by the client mode processor is communicated to the server mode processor, or how results are returned. CLIP is concerned with mappings of calls at the conceptual language level, not at the communications protocol level, a point we shall return to in a moment. In the examples cited earlier, of generic procedure libraries and mixed language programming in the same host environment, such communications may be routine, even trivial. This was a deliberate design decision at the time that the scope of the project was defined: knowing that important application areas needing the standard existed in single environments containing no significant communications problems, it was felt that the CLIP standard should not run the risk of unduly restricting language-independent procedure calling because of constraints made necessary in contexts where communications limitations were greater. There was no need to go any lower than the computational levels defined earlier.

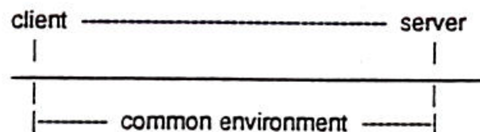
This brings us to the relationship between CLIP and RPC.

Remote Procedure Calling using OSI (RPC)

While this work was going on, ECMA was working on its own standard, already cited, for remote procedure calling, where the client's call of a procedure and the return of the results by the server are communicated through OSI protocols. That this was happening gives evidence of the need for a standard definition of procedure calling, but (as is all too common in the standards world) it was done without, for a time, any apparent awareness of the project in WG11. In the tradition of the area, the RPC folk went ahead and defined their own, based on their perception of need; but also a specific model of procedure calling in an OSI environment. When the two projects later came together in ISO (though in different committees and working groups), this provided valuable source material for the development of CLIP, but it was apparent that, being predicated on the client and server communicating by OSI protocols, in CLIP terms it was constrained by limitations at the communication level which are independent of the language level which CLIP addresses. Another way of putting it is that RPC is driven by OSI rather than by procedure calling.

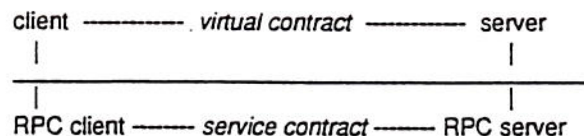
In principle CLIP and RPC were and are not in conflict; the task remaining for both WG11 and the equivalent OSI group, SC21/WG6/RPC, is to ensure that they can coexist without conflict. It is vital that CLIP is not defined in a way that would create problems for the RPC/OSI community, since RPC applications are potentially just as important as procedure library and mixed language applications - indeed can complement them by extending their range. However, as mentioned earlier it is equally vital that CLIP itself not be constrained by communications considerations. These matters are being handled through consistent use of definitions in a common Interface Definition Notation (IDN) employed in both RPC and CLIP and, because of the need in both for definitions of parameter datatypes, in CLID also.

The model that has developed can be shown in schematic form in the following way. In the purely CLIP situation, the model is simply:



where the common understanding above the line is CLIP's concern, and how the common environment is provided to support facilities using that common understanding is below the line and is not CLIP's concern.

Since RPC is dependent on OSI protocols, it is inevitably involved with the representational and implementation levels, at least to the extent of requesting lower level OSI protocols, so the model becomes this:



in which, as before, CLIP is concerned with what is above the line and RPC is concerned with what is below the line. What matters is to ensure that CLIP virtual contract and the RPC service contract match, and share the same view of what it means to call a procedure, to pass a parameter, and to pass back results. The two are well aligned now, and the only difficulties in maintaining that are likely to prove procedural rather than technical.

Further possible language independent standards

Those are the actual projects in train at present, but there are numerous other possibilities for generic language independent standards. Input-output is an obvious example: for text I/O at least, it is ludicrous that every language definition has to have long chapters each reinventing, with variations, that particular wheel. Yet old attitudes and habits of mind die hard; not very long ago someone suggested to a language standards group modelling their I/O on what was in another language standard, and this was rejected as unthinkable!

Other examples are file handling, array handling, parallel processing, and exception handling (though not "event handling" in general, since this would cover process control facilities and would qualify as "functional" for most existing languages). Some of these have already been the subject of some study, though none have yet reached the status of an official project.

Much may depend on how well the standards already described are accepted when they appear; if they are successful, the benefits of language independent definitions for the many areas of commonality will be more widely appreciated. Much will depend upon us, the users, insisting that these standards are indeed used, and incorporated in products. We are the ones, after all, who most suffer the difficulties and costs of incompatibilities.

Concluding remarks

It is clear that we are still on a steep section of the learning curve for dealing with language independent facilities and standards, both for those working on the language independent projects and for those in the various language communities. As we have seen, traditionally the language communities have been largely independent and disjoint, doing things their own way and talking only to their own kind - and those who move to the cross-language projects cannot always cast off immediately all the inbuilt assumptions from their respective backgrounds, whether they be from Fortran, Cobol, Pascal, C or whatever. It is hoped that this paper will help to clarify some of the issues and help readers to climb a bit further up the learning curve.

Acknowledgements

Although revised and substantially rewritten for the purposes of this paper, a substantial part of the material used above was originally drafted for a WG11 document, reference N194R, while much of that used in the discussion of LIA was originally drafted for the forthcoming book *User Needs in Information Technology Standards*, edited by Cliff Evans, Brian Meek and Ray Walker and due to be published by Butterworth-Heinemann by the end of 1992. The author is grateful to various members of WG11, in particular Willem Wakker and Brian Wichmann, for their helpful comments. An earlier version of the paper in its current form, for which the author is solely responsible, was presented at the DECUS Symposium in Solihull in May 1992.

References

Standards and drafts

ANSI/IEEE Std 754:1985, standard for binary floating-point arithmetic

ECMA-127, RPC (Remote procedure call using OSI), final draft 2nd edition, January 1990

ISO/IEC TR 10167, Guidelines for the preparation of programming language standards, 1991

ISO/IEC JTC1 CD 10967, Language Independent Arithmetic - Part 1, 1991

ISO/IEC JTC1 CD 11404, Common Language Independent Datatypes, 1991

ISO/IEC JTC1 SC22 WG11 document N295, Common Language Independent Procedure Calling Mechanism, 1991

Other references

[Kahan 1992] KAHAN, W., Analysis and refutation of the LCAS, *Sigplan Notices of the ACM*, Vol 27 No. 1, pp 61-74

[Meek 1990] Meek, B.L., Two-valued datatypes, *Sigplan Notices of the ACM*, Vol 25 No 8, pp 75-79, August 1990

[Payne, Shaffert and Wichmann 1990] Payne, M., Shaffert, C. and Wichmann, B., Proposal for a language compatible arithmetic standard, *Sigplan Notices of the ACM*, Vol 25 No 1, pp 59-86, January 1990

[Wichmann 1990] Wichmann, B.A., Getting the correct answers, National Physical Laboratory Report DITC 167/90, June 1990