User-Defined Types in a Polymorphic Language

David M. Harland
Department of Computer Science, University of Glasgow, Scotland

This paper will briefly review the common view of data types and summarize the various methods of deciding type equivalence (i.e. name equivalence, isomorphic equivalence and occurrence equivalence). After criticizing the essentially lexical nature of these mechanisms, a new rather dynamic view of types will be introduced and with it a very simple equivalence mechanism. This dynamic view of types will then be shown to be essential if types are themselves values in a language, as we argue they must be if that language is 'polymorphic' while supporting data protection via 'type constancy', especially if user-defined types are to be freely abstracted over, as is demanded by the principle of procedural abstraction. This view of types and type equivalence, when combined with higher-order functions and general type polymorphism makes 'abstract data types' almost trivial, and, as we shall see, it does so without interfering with the classical concept of block structured scope. This, in turn, supports and encourages 'modularity' in software.

1. INTRODUCTION

In this paper we shall outline the essential requirements of a polymorphic language which supports data protection via cell constancy, particularly with respect to the notion of type which it must embody. We shall discover that commonly encountered type-definition mechanisms are inadequate in a language where types are themselves manipulable. User-defined types are chosen as the vehicle for the discussion in this paper simply because they illustrate the dilemma that arises with respect to type equivalence when traditional, essentially static, notions of type and type equivalence are carried forward into such a highly dynamic language.

The use of strong typing has revolutionized programming practices, and greatly enhanced software reliability and data protection. Modern languages therefore try to encompass as large a range of data types as possible. Although it is clearly impractical to provide a programming language which directly supports all the data formats likely to be needed by its future users, it is generally acceptable to provide the users with a built-in mechanism for combining existing data types so as to form new composite, or structured, data types. Of these user-defined types the *record* is perhaps the most often encountered such mechanism. We shall concentrate upon the record mechanism in this paper. This will provide a focus for our criticisms of user-defined types; we have no quarrel with the idea of a record as such.

In a language which is strongly typed the question of *type equivalence* for such new data types naturally arises: when are two records considered to be of equivalent type? This is the problem which we wish to highlight here, and, after briefly reviewing existing strategies, we shall outline a new approach to user-defined types which we feel offers several advantages, not the least of which is simplicity. We shall then explore the consequences of having types as values, particularly with respect to side-effects affecting user-defined types.

2. TYPES AND POLYMORPHISM

Before we discuss types as such, and user-defined types in particular, we shall set the scene by outlining a form of polymorphism which we find attractive.

Polymorphism, as its name suggests, means that something will assume different forms at different times. In relation to programming languages we see its role as being an optional constraint on the form that a storage cell can assume. In particular, we assert that whenever a cell is created (or allocated for use, if cells are reused within their lifetime) it assumes a particular degree of type- and value-constancy. Strachey has argued[1] that constancy is a property of a cell. We agree. Our kind of polymorphism is therefore related to storage cells. The particular constancy requirements of a cell constrain the type of value that it can hold, if it is type-constant, and whether or not it can be updated, depending upon whether it is value-constant.

Taking these two degrees of freedom there are four options. A cell may be declared to be type-constant, i.e.

let *x int* := *E*

where *E* is some suitable expression. This cell will only hold integers. Similarly, should it also be value-constant, we have:

let *x const int* := *E*

which, once initialized cannot be changed.

The remaining two forms are, therefore, a constant cell which is not type-constant:

let *x const* := *E*

which, once initialized cannot be updated, but has the advantage that it can be initialized with anything at all; and, ultimately, the truly general purpose storage cell:

let *x* := *E*

This can be initialized with any value of any type, and can thereafter be updated by any other value of any other type.

Naturally we can perceive among these forms the traditional LISP or APL variable, which is polymorphic, and the Pascal cell which is more or less equivalent to the type-constant cell, apart from the fact that Pascal does not have initializing declarations; and its constant-system which is effectively the type-constant cell (although the implementation will be different).

By making the choice of the desired degree of constancy explicitly available to the user we enable him to choose that delicate balance between expressive power and data

CCC-0010-4620/84/0027-0047 $05.00

security, and let him choose it for each variable he declares to suit its application. We have found this to be most useful, very natural (more so than any single form of constancy on its own) and quite adequate for our desire to write truly general purpose, yet secure, 'software tools'.

To write generalized routines, 'abstractions', we must also enable parameters to create cells with these forms of constancy, or else we cannot abstract over their in-line equivalents. This is the principle of declaration correspondence.[2] It demands precisely what we want. We agree with it, and adopt it wholeheartedly. We therefore have a full range of equivalent parametric declarations, one for each of the above in-line declaration forms.

Now, it would not be very good if we could *only* specify the types in these type-constancies by embedding some particular type-name. To be able to abstract over these constancy specifications we must be able to use arbitrarily complex *expressions* at these points, otherwise constancy specifications become 'frozen in', being literally expressed. To achieve this freedom of expression we need to be able to manipulate types themselves. Classically, that is mathematically, such manipulation is embodied in the process of 'computation'. Computation is therefore the corner-store of computational science. Computation is generally agreed to be concerned with the act of evaluating expressions to yield values. It is in this light that we must examine types. The aim of this paper is, therefore, to justify the elevation of types to being fully manipulable values in a language (see Section 5) so that we may abstract over them (see Section 7). The form of polymorphism introduced above turns out to have quite a lot to say about types.

3. TYPES AND TYPE EQUIVALENCE

In language manuals the notion of type is usually skirted over as being intuitively obvious and therefore not requiring an elaborate exposition. Detailed discussion is usually reserved for enumerating the particular types which are included. This is unfortunate, because the precise meaning of type, which lies at the heart of most programming systems, is far from universally accepted. Indeed, many people use the term but introduce it in apparently different ways. Perhaps most popular, among the more rigorous approaches, are the 'axiomatic' and 'algebraic' (see Ref. 3 and references therein) type models; whereas the more abstract, but ultimately more powerful, lattice-theoretic models of Dana Scott[4] are to be found deep within most approaches, so the range is perhaps narrower than it seems.

Once we have decided what types actually are, we must examine how they are to be exploited in a programming language. It is widely accepted that the use of types enhances program reliability, by increasing data security—only sensible operations are allowed on data of different types. Most language designers demand that compilers undertake to check programs for such senseless operations, as far as is possible, and it is to this 'type checking' that we must turn now.

To be able to undertake type checking we must determine when two types are 'equivalent'. But what does type equivalence actually mean? Whatever it means, it must be defined in terms of the type model adopted, and so it will tend to be different in languages which have different type models.

There are three principal forms of type equivalence criteria, all of which concentrate on the form that a type definition takes within the *text* of a program. They are therefore essentially 'static' equivalences, since they are based upon syntactic analysis.

Perhaps the simplest is that based on the equivalence of type-names. Namely, two type definitions which have the same name are deemed to be equivalent. The obvious danger is that the detailed structure of such composite types may well be completely different. An obvious refinement, therefore, would be to say that only occurrences of a name within a given scope level (where that name has a unique meaning) are equivalent.

Secondly, there is a structural equivalence strategy. This states that any two type definitions which give rise to the same data format (after macro expansion of embedded type names, etc.) are deemed to be equivalent, irrespective of their user-coined type-names. The obvious drawback here is, of course, that data that are distinct to the user are all the same to the system, and so it is quite possible for data to inadvertently migrate, because they have 'the right shape', into places where they are essentially meaningless. This seems to be somewhat counterproductive, since the point of a data type system is to make data secure.

A third mechanism, usually called occurrence equivalence, states that each occurrence of a type definition in a program's text gives rise to a distinct data type. Thus the equivalence is entirely dependent on the textual form of a program. This suggests the opposite of the isomorphic equivalence, that is that identically formed types are distinct.

Obvious complications and side-issues affecting such equivalence schemes are the precise ordering of components of a structured type (with regard to the isomorphic equivalence scheme), and the so-called 'anonymous types' (in the case of name equivalence) which have no type-name as such.

4. A CRITIQUE

One major criticism of the above mentioned type equivalence mechanisms, which are to be found in many of today's programming languages, is that they treat types as 'second class' objects. They discriminate against types, as they do against procedures, by making them denotations, and not values in their own right. That is, they give types names, making them essentially lexical objects. By discriminating against types, so that they cannot be evaluated via arbitrarily complex expressions, they become entirely known statically, making them amenable to complete analysis during compilation. Most languages designers regard this as an advantage, because it makes programs easier to check during compilation. Although we acknowledge that type checking is highly desirable, in principle, we feel that achieving this by discriminating against types actually complicates the programming language (as we shall argue below).

Given that most popular languages have such 'static' types, it is no surprise, therefore, to find that type equivalence boils down to either equivalence of the name given to the definition, or the lexical form of the definition, or the position of the definition within the text.

Discriminating against types, by removing the ability to evaluate them, is a slight against the teachings of Strachey, who felt that all things should have the same 'civil rights'. Although he did not actually consider types in this scheme, we feel that his approach would be to make them 'fit in' with the other things in a language. The popular obsession with wholly static type checking has, we feel, given rise to a veritable albatross sitting squarely on the shoulders of today's programming language designers, and it seems to be retarding the development of what we would regard as better languages. But how do we know when we have produced a better language? Two factors must be considered. A good language is a 'simple' one. A useful language is one which is well suited to the area of its application. A language can be judged pragmatically, to see how well it fits with the application in hand, and it may be judged in a more abstract sense to see if it is sufficiently simple. Measuring simplicity is not an easy task. If we distinguish between simplicity which arises from triviality, and simplicity which arises from a uniform structure, then it is clear that we must measure the degree of structure in a language in order to determine whether it is a simple language or an inherently complex language. We feel that the classical notion of computation, the evaluation of values from expressions, is the expressive structure that is required in a programming language. Anything which hinders unfettered expressivity, such as discriminating against types by forbidding their evaluation, complicates this structure, and therefore makes a language more complicated than it would be if its types were fully manipulable. It is in this sense that the struggle for wholly static type checking in modern programming languages is seen as being counterproductive.

5. TYPES AS VALUES?

Values are abstract entities, carriers of information, arranged according to their properties, granted by way of the operations available to them, into type spaces. Every value has a type; we shall argue later (see Section 8) that every value has exactly one type. Given a value, we should be able to deduce its type. To specify the type of a value we can 'tag' its internal representation with an internal representation of that particular type. There must be a unique tag for every type that must be represented. In terms of an implementation scheme, therefore, every value has a type-tag inseparably bound to it, the pair being passed and assigned as one.

Here we assert that not only should types be values in a language's universe of discourse, they should be 'first class' citizens in the language, just like all other values.

This assertion contains two proposals; that types should be values as such, and that these should then be 'normal' values (in the 'clean' sense of Strachey). Let us examine each aspect in more detail, to discover why, if indeed it is so, that types should be freely manipulable.

We could, strictly speaking, adopt types as values but discriminate against them to prohibit arbitrary type-valued expressions. If we resist making types fully manipulable values, what are the consequences in the light of the above type-constancy options?

First we would have difficulty specifying arbitrary forms of type-constancy; we would be able to specify the desired type literally but we would not be able to evaluate it, as the result of a function, say.

Secondly, we could find ourselves in the situation of having a polymorphic variable and wanting to use its content value but of not knowing the type; how might we find out? One way would be to employ type-testing predicates, such as *IsInt* etc. to test the type of a value, without directly accessing the type. This might well be acceptable in languages with a fixed number of types (and hence predicates), but in a type-extensible language such predicates are far too restrictive; they require that the user already have some notion as to the type that it is likely to be. Predicates are unwieldy, in a polymorphic system, as a great many type tests become necessary every time a polymorphic variable is accessed in a type-sensitive operation.

It is much simpler, we feel, if types are values in their own right and we have a *typeof* operator which directly yields the type of a value (by examining its tag). Consider, for example, a routine for writing out its parameter along with its type: which would be easier? To have a long test sequence of predicates? Or a *typeof*-like operation? As suggested above, predicates are actually impractical in a type-extensible language. Where do all the predicates come from for these user-defined types?

Thirdly, unless types are values how can we sensibly parameterize with them and, most importantly, how can we abstract over them?

To fully be able to exploit our form of polymorphism (as opposed to other forms of polymorphism, see Section 11) it seems that types must be values. But should types have the same 'rights' as other values?

The term 'first class', as applied to values, simply paraphrases the case made by Strachey for the principle of data type completeness (as it is now widely called). This means precisely what is says: all values, whatever their type, must be able to pass into and out of routines, be assigned freely and stored away in data structures without restriction, so long as there is a suitable receptacle to receive them (in a strongly typed system). That is, no bar is placed on their movements on the basis of their type as such, unless it is a type-constancy specification.

If completeness is upheld when we make types values in their own right, allowing arbitrary expressions of type type, then we naturally have to forsake the hallowed complete compile time checker because we have removed the very prop which made it possible. That is, the type of the result of an expression may not, now, be known statically. Similarly, the type specification of a variable's cell may not be so-fixed, but be evaluated dynamically, so that different incarnations of a given variable will give rise to differently typed cells at run time. This is polymorphism at work. It permits the user of such a programming language to write truly general purpose programs, the much sought after 'software tool', without the 'verbose static' nature of today's generic packages.

6. A PROPOSAL

If we accept, for the rest of this dicussion, that our programming language has types as values, and that its variables and cells for parameters of routines, and similarly those within its composite types may dynamically determine their type requirements, making them polymorphic, then we can proceed to outline a further type equivalence mechanism. One which it is felt is superior to the above mentioned schemes.

A data type is defined here to be an abstract value-space, occupied by values having that type, and for each such data type there is an entry in the space containing values of type type. In traditional parlance a 'type expression' is really what we shall call the 'type definition'. Since types are themselves values here it is obvious that *any* expression yielding a type, whether it is a new type or an existing one, is technically a type expression. We shall regard each 'type definition' as being a call to a 'generator' which, when called at run time, yields a new type value and has the effect of creating a new data space for values of that type to reside in. This new type value, yielded by the definition, is presumably stored by the user in a suitably typed variable or data structure for subsequent use. Note that there is no type-name as such, simply a value of type type retained in a location in memory, it is the storage cell which can be named.

A user-defined record type, forming an expression of type type, could be specified as follows (using Wirth's meta-syntax[5]):

"*record*" "(" [*fields*] ")"

where the fields are given by:

field.spec {";" *field.spec*}

for a field specification:

literal {"," *literal*} [":" "*const*" *expression*]

The optional expression clause in the field specification, if given, is the type constancy of that group of field cells; if omitted then the field cells are completely polymorphic—constancy is a property of the storage cell itself.

It is important to recognize that, in contrast to popular practice, the various field literals are *values* for selectors which can be used to index into incarnations of the corresponding records. The field selectors naturally have full 'civil rights' in the language, so that they can be stored and passed around with complete freedom. Consequently, it is now possible to have generalized accessing operations, which evaluate both the record desired and the field to be selected, namely

search (*ptr*[**if** . . . **then** *left* **else** *right*])

for an incarnation of a record given by *ptr* which has fields *left* and *right*.

Notice that there is no restriction on the form of the expression which gives the type-constancies of the field cells of a record definition. In particular it is feasible to have truly recursive record definitions, in the sense meant by Hoare,[6] without the need for explicit pointer types. Thus:

let *tree* := *record*(*item* ; *left*, *right* : *tree*)

which would serve for the above accessing example. Naturally, in the implementation, when defining a record only the general shape would be recorded, and when a particular incarnation is created space is allocated in the Iliffe-style, with records pointing at records. To terminate the recursion in the allocation we need an 'empty' record of that type or a special universal 'nil' value. We prefer to have an empty value of the right type than a generalized nil. The empty incarnation would be created as follows:

let *empty const* := *tree*[ ]

and subsequent real incarnations could use this, or create them explicitly, since data structures are initialized on creation:

let *root* := *tree*["*TOP*", *empty*, *empty*]

Of course, we would have to have a predicate for testing trees for being nil, unless we always want to test for equality with an empty one explicitly. We therefore have a 'nil' predicate which is polymorphic in that it takes any type of data structure and reports whether or not it is a trivial value.

Now we must move on to examine the behaviour of type definitions in more detail, particularly with regard to being values and hence being able to be abstracted over. As we shall soon discover abstractions come in various forms.

7. ABSTRACTING OVER TYPES

Obviously, as we encounter multiple in-line occurrences of a similar type definition we find that we get a number of distinct new types, irrespective of their internal structure. If we recall that every value has its type inseparably bound to it, like a tag, then we can always tell the type of any value encountered during execution. It is therefore impossible to confuse values arising from similar definitions. This prevents inadvertent migration of values; the problem with values from similar definitions encountered by the static isomorphic equivalence scheme. Our equivalence system is clearly not name equivalence (there being no type names to compare). It is a form of occurrence equivalence. But it is not the same as the above occurrence system, as we shall discover shortly.

We note in passing that there would be no problem with the Pascal-like 'anonymous type', namely

var *X* : (*red*, *green*, *blue*)

as this forms the basis of the type definition clause, and we now have no type-names anyway. The problem for Pascal is of course, what does this mean to a name equivalence mechanism?

The principle of procedural abstraction requires that any in-line syntactic clause may be 'abstracted over', to form a routine body, so that calls to it are placed in the program instead of each such occurrence. It further demands that this be so with the minimum of reconstruction of the program and *without changing the meaning* of the abstracted clause. The principle of procedural abstraction is very important, as it provides the means by which large complex programs are broken down into numbers of individually simpler sections of program. Any implementation of the principle which requires the user to rewrite large parts of his monolithic program, in order to abstract out common parts, is counterproductive. Any implementation which discriminates against certain syntactic clauses or data types, so that not all constructions can be abstracted over, is similarly undesirable. It is important, therefore, that the principle of procedural abstraction be carefully and completely applied. To be fully implemented procedural abstraction requires that data type completeness and declaration correspondence be completely supported, otherwise there are values, types, declarations and operations which are context dependent, which then hinder generalized abstraction. But these are not by themselves adequate prerequisites for unfettered abstractions, it is essential to have type polymorphism so that parameters can abstract over a range of in-line variablecharacteristics used by a common algorithm; and polymorphism, when combined with truly flexible data structures, particularly lists, gives the heterogeneity seen in parameter sequences (so that parameterization may be fully abstracted).

When combined, all of these generalizations enhance expressive power, and hence aid abstraction. To demonstrate that this kind of power is not only highly desirable but strictly necessary we need look no further than the classic *write* statements. In Pascal, for instance, this inbuilt routine takes advantage of variable-length parameter lists, and if we regard this list as a data structure we see that it exploits flexible heterogeneous lists, as it takes a variety of parameter types. To be able to abstract over the *write* statement we, as users, would need to have that same power, at least. In Pascal we do not have it. Why then is it built into the *write* statement? Because the language is totally useless without it. If it is so obviously useful why is this kind of expressive freedom denied to the users for their own routines? We should, given adequate adherence to procedural abstraction, be able to write:

```
let p := procedure ( x ) write (x)
```

which takes in any value as a parameter and passes it on to *write*; if this was in fact a list then this would be passed on too. In a particular language there would have to be some convention, of course, for exactly when a given actual parameter list is passed as a list and when it is stripped up for initializing a sequence of formal parameters, but this is a trivial matter. The point is one of principle. We should be able to abstract over any syntactic clause, including not only the user's own abstractions but those which are built in for him.

Naturally we should then be able to abstract over types too. To be able to abstract easily we make things values, so that we can 'compute' with them. This is exactly why we insisted that types be proper values. A type definition is a clause, an expression, which yields a *new* type. As we have already seen there is some contention as to exactly when and how type definitions make these new types. However this is finally answered, it is apparent that it must be consistent with the notion of abstracting over type definitions.

If we abstract over a record type's definition, so that:

```
let x := record ( x1, x2 : int )
```

becomes:

```
let f := function () record (r1, r2 : int )
:
let x := f ()
```

the type, as a value, is passed out of the function almost immediately that it is created. It is therefore manipulated indirectly, via expressions of type type. The type has no name, it is a value. Any expression which yields a value of type type is therefore a 'type expression', irrespective of whether it actually creates a new type. It is on a par with 'integer expressions' and 'function expressions'. Types as such are only really strange objects when they are 'special', that is when they are *not* actually values. When they behave like normal values types lose their mystique.

We also see that whereas the field literals were originally in the same scope level as the variable *x* itself, now they are within the body of the function, and so are unavailable beyond it. All is well, however, because fields are values in their own right, and so, if the fields cells are to be accessible beyond the body of the function, the field selectors, like the new record type itself, must be passed out of the function. This is most easily done if the language supports both a parallel assignment operation and lists as a data type. Thus, we might write:

```
let x, f1, f2 := f()
```

where the abstraction is now:

```
let f := function ()
     begin
              let r := record (r1, r2 : int)
     →        list [r, r1, r2]
     end
```

which explicitly builds a list value as its result and employs the notion of a block-expression to yield it.

If the field selectors were not first class values in the language this would not be possible, and so it would not be practical to abstract over the record definition, making a restriction on the abstraction mechanism in contravention of the principle of procedural abstraction.

We should, applying this principle, be able to abstract over *multiple* in-line similar type definitions, making a routine which does the same job, and call it wherever it is needed, without adverse effects. Thus:

```
let a := record (a1, a2 : int)
:
let b := record (b1, b2 : int)
:
let c := record (c1, c2 : int)
```

should (ignoring the need to pass the field selector values out too) be equivalent to:

```
let f := function () record (r1, r2 : int)
:
let a := f()
:
let b := f()
:
let c := f()
```

To be so we find that each call of the routine, which executes the all-important type definition clause, needs to create a distinct new type to be returned as its result, or else we find that we have produced a form of isomorphic equivalence for this construction as opposed to the form of occurrence equivalence that we were describing in the multiple in-line format. Our conclusion, for distinct types from each call of the routine, is in direct contrast to the static view of types described earlier, where the same type would arise in all three schemes. If we had adopted any of these earlier schemes we would find, as we shall see soon, that side-effects could give rise to wholly unacceptable situations which undermine the logic of their strategies. Our new, essentially dynamic, view of types can, however, handle such side-effects sensibly and simply. Before we move on to explore this further, it is necessary to outline a corollary to this interpretation of the needs of procedural abstraction.

We view loops as being equivalent to a (dynamically determined) number of in-line occurrences of the loop body, as yet another form of abstraction. So, similarly, any loop which includes a type definition must give rise to a distinct new type, otherwise the effect will be different from a number of in-line cases. The above example would therefore be equivalent to:

```
let a, b, c := for i := 1 to 3 eval record ( x1, x2 : int )
```

This is entirely consistent with procedural abstraction, as the effect would be the same if we abstracted over the loop body to form a routine and then called that as the loop body. This demonstrates too that looping constructs and abstraction are equivalent. This fact is encouraging, since if it were otherwise we would have to recognize that something was seriously wrong with the design.

The acid test, as indicated above, concerns the response to side-effects. Since the type specification of a variable, in our assumed polymorphic language, is evaluated dynamically we might get differently typed variables at different times from the same piece of program text. The same facility applies for the cells which form the fields of records. Side-effects in the environment of a record's type definition, giving rise to differently typed field specifications, must not have adverse effects on the type system, more particularly the type equivalence system. In short, these too must be orthogonal. Consider, for example, the following program text:

```
let x := 999
:
let f := function () record (r1, r2 : typeof x)
:
let x1 := f() [42, – 1]
:
x := 'a string typed value—it's a polymorphic cell'
:
let x2 := f() ['now they', 'are different']
```

where individual types produced by this definition have different internal structures, simply because we have exploited the generality offered by our programming language. Whereas our new language, with its dynamic view of types, would be secure because each incarnation would be a distinct new type, a language in the more traditional stable, having a static view of types, would find that a single lexical occurrence of a definition with the same lexical template and the same type-name in a given scope gave rise to completely different structures. Would we still hold that they were equivalent under one or other of the original trio of equivalence schemes? In fact this particular problem never arises in traditional languages since the generality or expressive freedom demonstrated here is not possible: it is forbidden because types are discriminated against, making it impossible to abstract over type definitions and impossible to have side-effects on type specifications for variables and record fields. In short they do not support the polymorphism which highlights the issue. Some, of course, might be only too relieved to hear that such 'side-effects' are forbidden, and would forbid more sources of such problems if they could. The main difficulty there, of course, is that computation in traditional languages is based upon the act of committing a side-effect, and such restrictions, if they were introduced, as they are for types, merely serve to undermine a language, not enhance it.

8. A NEW FORM OF TYPE EQUIVALENCE

We have seen that a 'type definition' clause is in reality a call to a nullary operator, and that to be consistent this is invoked dynamically every time it is encountered during the 'flow' of the program.

In our model, therefore, types involve 'spaces' which are generated dynamically within which values of that newly defined type reside. In fact, to avoid all manner of problems regarding reflexive types we must demand that these spaces be complete continuous lattices, as described by Scott in his data type model. The types themselves are, however, really algebras. Each type, considered as an algebra, encompasses a family of operator functions, in addition to the carrier space for the values of that type. It is these operator functions which provide the degree of interpretation required to provide the abstraction inherent in the new type. To be acceptable to Scott's thesis these operator functions must be continuous functions. This, then, is the type model. Our interpretation differs from the popular view only in that these type spaces are generated dynamically, with the consequence that a given lexical type definition generated a whole range of types, and not just one.

Now we come to the vital question of type equivalence. We must specify the equivalence in terms of the type model, so we can state straight away that it must be an algebraic equivalence. To be equivalent two algebras must have equivalent carrier spaces and equivalent operator families. How might this work in practice? First, since it is impossible to prove two arbitrary functions to be equivalent in the general case we must restrict ourselves to the decidability of particular cases. In implementation terms we can compare function closures, to see if they are the same function, but this is about the best that we can do, since we cannot prove that two different closures are equivalent. However, as the language being discussed above only permits the user to specify the form of the values which will reside in the new carrier space, and does not allow him to 'attach' the various operators which go along with it, the implementation has no idea which closures are relevant to which type (although this situation might well be remedied in the near future). All that we can do, therefore, to determine type equivalence in this algebraic framework, is to compare the carrier spaces. To be equivalent types these carrier spaces must be equivalent. The best that can be achieved when comparing a pair of spaces is to say that they are equivalent up to isomorphism. For infinite spaces this might prove rather difficult. All that we can actually do, in implementation terms, is to decide whether the two types arose from the same invocation of a particular type generator. This is easiest done by a tag-equivalence; each value being tagged with some representation of its type, with a new tag being created on every invocation of a type generator. Naturally, therefore, we advocate the use of a tagged architecture[7] for implementing polymorphic languages.

Type equivalence is, then, a matter of tag equivalence in an implementation in which machine support is provided for tagging and type checking, this being the closest one can come to a complete algebraic equivalence (with or without the closure-tests for operator functions).

In order to achieve this, we must ensure that every value has a type, as seems natural, with the further constraint that a value has exactly one type. This merely reflects the fact that any given value can exist only in one type space [see discussion in Ref. 8], although it may well be used in other data structure spaces, as a component. In the same way as we use the nullary generators to bring new types into being, we can, by a similar argument, employ the generators for these new spaces themselves to create new values of those types. This space-population commonly occurs during the definition of the new type itself, as in the case of scalars, or incrementally during execution, in the case of a structured type. In an earlier paper[9] we demonstrated the manner in which enumerated types come into being, and showed how the values created of such types may be granted the full freedom of 'first class' status; indeed, they behave much as do the record field selectors described herein. As noted above, the ramifications of values being endowed with exactly one type have been fully discussed elsewhere,[8] with reference to the popular notion of a 'subtype'. In this paper we shall concentrate on the structured types.

Having finally discovered a type equivalence scheme which is adequate for the highly dynamic view of types required by the combination of type polymorphism and type constancy, where types are values in their own right, we shall resume our exploration of its consequences.

9. ITS CONSEQUENCES

So far we have looked at the influence of having types as values in a language, and have examined the possible consequences of side-effects on expressions for general cell constancy. We have seen that this, within type definitions for structured types, requires that we view types in a highly dynamic way, much more so than is traditional. We then discovered that this approach leads to a clear view of type equivalence, and is wholly consistent with the expressive power inherent in the principle of procedural abstraction.

Above we saw how a dynamic view of types requires that a given type definition clause give rise at run time to distinct record types with differently typed field cells. Now we shall move on to show why it is necessary to allow different *incarnations* of a *given record type* to have differently typed field cells.

If we consider further the nature of cell creation in relation to the evaluation of the cell's constancy clause we discover that it is necessary to view composite types independently of the field constancy requirements, rather like divorcing the length and component type from the array type, by making them attributes of individual values of that single structured type.

We have already seen that in our polymorphic language storage cells can have various degress of constancy, and that this constancy is evaluated as the cell is created. Consequently, as cell creation (or allocation) occurs dynamically, a variable's cell constancy may be different on different invocations. In a similar manner a routine parameter's cell, although type constant, may be differently so on subsequent calls of that routine, because parametric cells are created when the routine is called, not when it is created—routines are values in our language, remember. This, then, is the setting.

An identical argument concerning field cells of record types demands that the constancy attributes are evaluated each time that a record is invoked, not when the type is created. This is so because a record specification is really just another form of abstraction. In this case it is a means of creating a group of related cells. The fact that it also introduces a new type and makes available a number of selectors for accessing the individual cells within the group is quite another matter.

Each invocation of a record value from an established structured type is effectively an abstract form of creating an identical group of in-line variable declarations, so, to preserve side-effects, we clearly must evaluate cell constancy for field cells when a data structure is actually created.

Thus we might write:

let *stype* ⁚= *record* (*a*,*b*)
:
let *p* ⁚= *stype* [2,3]
:
let *q* ⁚= *stype* ['*A*','*B*']

where the fields are polymorphic by way of there being no cell constancy at all. This, however, is not very secure. We might prefer some degree of type constancy:

let *stype* ⁚= *record* (*t* : *type* ; *a*,*b* : *t*)
:
let *p* ⁚= *stype* [*int*, 2, 3]
:
let *q* ⁚= *stype* [*char*, '*A*', '*B*']

In practice this might turn out to be rather verbose if we vary the type infrequently, so we might prefer to achieve it via a side-effect on the definition cell's constancy expressions:

let *t* ⁚= *int*
let *stype* ⁚= *record* (*a*, *b* : *t*)
:
let *p* ⁚= *stype* [2,3]
:
t ⁚= *char*
let *q* ⁚= *stype* ['*A*', '*B*']

which involves fewer arguments to passed each time.

10. ON ABSTRACT DATA TYPES

Recent years have seen much literature published on the topic of 'abstract' data types. Many proposals have adopted and extended the SIMULA 'class' concept,[10] with its property of environmental retention and the exporting of local names for use outside. Proponents of data abstraction via class-like objects argue that the classic block structured scope rules of Algol-60 are inadequate for information hiding activities. It is precisely because the (extended) class feature selectively exports its local objects that this scoping restriction can be overcome.

Any language which adopts this programming methodology must, therefore, abandon the very simple and elegant scope rules of Algol-60. This seems to be a heavy price to pay just for abstracting over types. It is, in fact, really an unnecessary sacrifice because a little thought reveals that it is not the classical scope rules of Algol-60 which are defective but the proliferation of 'second class' values in languages which prevents information hiding operations. If types are full values and can be passed around then scope becomes irrelevant as an issue.

Here we have already shown how to abstract over a type definition, and we did so by employing functions in a clean language. We could just as easily have written a function which, in addition to defining a new type, associates some access functions on it and passes these values out to be used in the calling enviroment. Details of the type's implementation are invisible to the external user, as required. To do this, though, as can be seen from the example, we must be able to pass routines around, including out of their defining scope, in such a way that they retain their free environments. Although a number of the Algols have experimented with first class functions, most have also imposed 'dangling reference' restrictions on passing them around. A non-LIFO storage management facility, such as is required by first class routines with environmental retention, is easily implemented in a heap, the management of a heap, in turn, is greatly simplified if a tagging scheme is employed, and this, it turns out, is exactly what is required by dynamic type checking in a polymorphic system where types are fully manipulable values. Clearly these advanced aspects of language design complement one another well; this is encouraging.

Notice that we achieve our goal by passing values out of a function, not by exporting names from special class-like blocks. We have therefore unified, rather than extended our concepts, merely by adopting Strachey's notion of 'first class' values as a fundamental design tenet.

Consider the traditional 'stack package' example:

```
let new.stack :=
      function (s)
      begin
            let stack := vector size s value 0
            let sp int := 0
   →        list
            [ ! make top, push and pop !
                  function ()
                  if sp > 0
                  then stack {sp}
                  else 0,
                  procedure (x const)
                  begin
                     sp := sp + 1
                     if sp > s
                     then begin . . . end
                     else stack {sp} := x
                  end,
                  procedure ()
                  if sp > 0
                  then sp := sp − 1
                  else begin . . . end
            ]
      end
```

```
let top, push, pop := new.stack (15)

push (999)
let a := top () + 1
write (a)
push (a)
write (top ())
```

The actual data structure, although invisible to the user, is retained within the environments of the routines which manipulate it. It is these access functions which we are passing out.

A similar capability is available in Ada, with its 'packages' and 'private types', but in our view the addition of the entirely new concept of the package, and the extra scope-like private attribute for types, is a language complication, not a simplification, whereas by giving existing concepts (routines and types) a more uniform treatment (as values) we achieve the same end (if not even more so, as we shall argue later in this section).

Whereas this particular example creates exactly one 'stack', known directly to it, we might have reason to have an additional function which dynamically creates the actual data structures and returns them to the outer level, so that access functions can take as a parameter the particular 'stack' to be operated upon.

Notice, however, that if we choose to pass the stack itself out of the function which created it, then we really do need a polymorphic language in order to do this, otherwise we could never return such a 'stack' value to the calling environment—there could be no suitable typed receptacle in that environment unless we export the type itself . . . and to to this, in a value-oriented language, we would need types as values, and hence reacquire polymorphism.

Furthermore, observe that the stack in the above example takes advantage of the polymorphic nature of our language by being completely heterogeneous, enabling it to store up values of any type, all intermixed along its length. In order to handle this stack, we permit the parameter of the *push* operation and the result of the *top* function to be general purpose. The *sp* variable, however, is restricted to integer values (and if we exploited the 'subrange' constraints of Ref. 8 we could restrict it to integers in the range $1..s$ where s is the dynamically specified vector size).

The stack so created is itself invisible to the user of *new.stack*. In this particular case it was implemented as a vector, but it need not have been so; the user has no way of knowing. If a new type was actually created for the stack then this would only be manipulable via its 'operator functions', to use the terminology of the algebraic view of data types, the 'carriers' being hidden. If we were to provide a means of tightly binding these operators to the carrier space then we would have a more realistic user-defined type mechanism. In this way, whenever that newly created type is passed around it carries with it the operators defined upon its values. Something along these lines would seem to be the ideal 'abstract data type' mechanism. Clearly the type-equivalence system outlined earlier would be adequate for this, even when such types are themselves values.

Since we make the stack abstraction available without explicit environmental linkage we see that this is a truly 'modular' piece of software, perhaps being worthy of the term 'software tool'. No 'export'-list is necessary, the linkage is via parameters and function values, not via naming conventions. Seen in this light the contemporary drive towards techniques for enhancing modularity, which rely explicitly upon environmental linkage for their effect, must be seriously questioned: surely it is better to link software together by passing values around than by freezing in static naming linkage? In the event that value passing *is* ever adopted as the primary linkage mechanism (and surely it must, eventually?) then it is obvious that all values, of all types, including types themselves, must be fully manipulable.

11. OTHER POLYMORPHIC LANGUAGES

The polymorphism involved here is slightly different to that of the few other languages which have experimented with 'general purpose software' within the framework of a typed system. In particular we combine polymorphism with type-constancy for storage cells. This, as we have seen, has a quite revealing effect on our view of types themselves.

Milner has proposed a polymorphic language.[11] This is an applicative language in which updatable variables are alien, and so he does not need the degree of type-constancy that we invoke. Nevertheless, whereas his parametric declarations involve specifying a type, his in-line declarations do not, thus indicating that he does not consider the principle of correspondence to be important in his design. Furthermore, whereas his routine definitions are 'templates' that take varying formats of inputs, individual incarnations of these routines are very specifically typed. In particular heterogeneous data structures, such as our stack, are ruled out.

The language RUSSELL[12] has also explored type polymorphism, but the nature of its type system restricts 'type expressions with a particular signature' to being referentially transparent, thus hiding all of the consequences of type-oriented side-effects that we have discussed here. The designers of RUSSELL lament that they had to go to such lengths to achieve this transparency, as it complicates their language's scope rules.

Gladney has proposed a polymorphic language, which bears a certain similarity to that discussed here, but he rules out 'first class' types, commenting[13] that he 'has not yet found a program that he could not write just as easily without them'. Perhaps the above polymorphic stack is such a program?

We feel, therefore, that polymorphism as it is usually encountered, in the dynamically typed languages such as LISP, is too insecure, and that it should be combined with type-constancy for variables, where desired, while enabling polymorphism when it is desired. When combined like this we can achieve general purpose yet secure software.

12. CONCLUSIONS

It is suggested that the present uncertainty concerning the nature of type equivalence stems from the essentially static, or lexical view of data types, and that to sensibly determine the nature of type equivalence it is necessary to take a much more dynamic, value-oriented view of types.

To achieve this we require values of type type (which is highly desirable on general semantic grounds in any case), and general polymorphism on the type specifications of storage cells, in the form of variables and record fields. We find that our programming language becomes much simpler, and with the simplicity comes generality, and hence expressive power.

It has been shown that the dynamic view of type creation is consistent with the principle of procedural abstraction, and with looping constructs. It is therefore possible to abstract over type creation.

The new view of type equivalence has been shown to be different from previous mechanisms (name equivalence, isomorphic equivalence and occurrence equivalence), being a highly dynamic form of occurrence equivalence, and has been shown to be easily capable of handling the possibility of side-effects in type specifications for variable and field locations arising from the first class nature of values of type type. It is obvious that of the four methods discussed this is the only one which would suffice in such a powerful language, as all of the others would give rise to compatibility problems.

The fact that complete compile time checking is no longer feasible is not seen as important, since the prop upon which such checking was based (the discrimination against types in the language) is in itself undesirable, so has been removed as a matter of principle. In our view, in addition to the simplification of the language, the gain in expressive power more than outweighs any possible advantage to be had from compiler checks.

It is, we feel, important clearly to distinguish between complexity in a programming language and complexity in an application's algorithm expressed in that language. A language's users should only have to contend with the complexity in their application, they should not need to contend with additional complexity in the language that they employ; they should be able to rely upon a simple yet uniformly powerful language which does not actively interact with the design of their algorithms. It is therefore the onus of the language designer to provide a language which is both simple and sufficiently expressive. The complexity in an application is part of the application itself, this must be left to the user.

In such a scheme it is necessary to undertake dynamic type checking. This is most easily done via a tagged architecture. As noted earlier, however, this is not too exorbitant as it at first appears, because we also need a non-LIFO storage system in order to implement routines as values, where environmental retention is required, and a heap is the simplest such system. In a heap system, a garbage collection facility is essential, and with self-identifying data this is a greatly simplified process.

Dynamic type checking, in itself, is not necessarily the evil that it is often made out to be, but the fact remains, if it is necessary then the price must be paid. Type checks need only be performed when type-constancy has been specified, in the case of updating variables, or when operators manipulate values; when values are passed around within the polymorphic areas of a system they need not be checked. Naturally, where operations *can* be verified statically a compiler can plant optimized (non-checking) code sequences; a 'traditional' program can largely be checked in advance, only where the polymorphism is actually exploited by the programmer does a dynamic check become necessary.

Acknowledgements

I would like to thank John Jeacocke, Arthur Allison and Les Smith for various discussions during the writing of this paper, and Hamish Gunn for earlier co-operation on the implementation of a tagged architecture for such a polymorphic programming language.

REFERENCES

1. C. Strachey, *'Fundamental Concepts In Programming Languages'*, Oxford University Programming Research Group (1967).
2. P. J. Landin, The next 700 programming languages. *ACM CACM* **9**, 157 (1966).
3. D. M. Dungan, Bibliography on data types. *ACM SIGPLAN* **14**, 31 (1979).
4. D. Scott, *Data Types As Lattices, Lecture Notes In Mathematics* **499**, Springer-Verlag (1974).
5. N. Wirth, What can we do about the unneccessary diversity of notation for syntactic definition? *CACM* (November) (1977).
6. C. A. R. Hoare, Recursive data structures. *International Journal of Computer & Information Science,* 105 (1975).
7. G. J. Myers, *Advances In Computer Architecture*, Wiley (1978).
8. D. M. Harland, Subtypes versus cell constancy with subrange constraints. *ACM SIGPLAN* **17** (12), 65 (1982).
9. D. M. Harland and H. I. E. Gunn, Another look at enumerated types. *ACM SIGPLAN* **17** (7), 62 (1982).
10. O. Dahl, B. Myhrhaug and K. Nygaard, *SIMULA 67—Common Base Language*, Norsk Regnesentral, Oslo, Norway (1968).
11. R. Milner, A theory of type polymorphism. *CSR-9-77. Computer Science Report*, Edinburgh University (1977).
12. A. J. Demers and J. E. Donahue, Data types, parameters and type checking. *ACM 7th Principles of Programming Languages Conference*, p12 (1980).
13. H. M. Gladney, Personal Communication (1982).
14. H. I. E. Gunn and D. M. Harland, Degrees of constancy in programming languages. *Information Processing Letters* **13**, 35 (1981).
15. J. C. Reynolds, A simple typeless language based on the principle of completeness and the reference concept. *ACM CACM* **13**, 308 (1970).

Received March 1983