Types are not Classes

G.V. Cormack, M. Judd and A.K. Wright

CS-86-28

Types are not Classes

G.V. Cormack, M. Judd† and A.K. Wright**

1. Introduction

In contemporary language design, it is widely accepted that a data type is a set of values combined with a set of operations on those values. Such types are specified using a mechanism akin to the *class* in Simula 67 [Dah 70]. In this paper we argue that this notion of type is inflexible, and frequently imposes an unnatural structure on programs. Classes are particularly inflexible in defining *polymorphic* operations – operations that apply to more than one type.

As an alternative we propose a more primitive type structure in which the notion of *type* does not include a set of available operations: a type is merely a *type mark* and a representation. The type mark is used in selecting from a space of available operations; this space is organized by scope and visibility-control mechanisms that are independent of the type structure. The type structure described here has expressive power and abstractive ability that compare favourably with that of other languages. We have designed and implemented the language **ForceOne** [Cor 86, Wri 86] that embodies the type structure. In this paper we describe the shortcomings of class-oriented types, and how our type structure addresses these shortcomings.

2. Shortcomings of Classes

Our criticism of class-oriented types derives from the fact that whenever a new type is created, we must enumerate the operations that may apply to the type and include them within the type's definition. This requirement conflicts with our view that programming is the incremental process of building new operations in terms of existing ones. Furthermore, it is often not obvious to what type a particular operation *belongs*; sometimes we can make an arbitrary choice but in other instances no choice is appropriate. This problem is particularly apparent if the operation is intended to be polymorphic: how can it be defined within a type if it is meant to apply to many types?

2.1. Non-polymorphic Examples

Suppose that there exists the previously defined type `integer` which has the operations `+`, `-`, `*`, and `/`, but *not* `**`. If a program were to require the use of this exponentiation operator on `integers`, in many languages it would be necessary to modify the definition of `integer` to include it. Such a modification is likely to be infeasible, and is certainly undesirable, as it defeats modularity: `**` can be implemented easily using existing operations; access to the internals of the type `integer` is not required. It is possible to define a new type `powerable_integer` that has the operations of `integer` plus `**`, but `**` could be applied only to values specifically declared as `powerable_integer`.

Further limitations of class-oriented types arise in defining related types. In the code generator of a compiler, we may require two types `pointer` and `offset`. A pointer is the address of a storage location and an offset is the distance from one location to another. We wish to define the following six operations.

* Department of Computer Science, University of Waterloo, Waterloo, Ontario N2L 3G1 Canada
† Softech Inc., 460 Totten Pond Rd., Waltham MA 02254 U.S.A.

```
1.  + :  offset  × offset  → offset
2.  + :  offset  × pointer → pointer
3.  + :  pointer × offset  → pointer
4.  - :  offset  × offset  → offset
5.  - :  pointer × offset  → pointer
6.  - :  pointer × pointer → offset
```

First, we note that there are three definitions for each symbol but only two types: there is necessarily an overloading problem if each type is allowed to contain one definition of each. Even if we rename each operation to avoid this conflict (an unacceptable concession, in our opinion) a fundamental problem remains: to which type does each operator belong? Only the first properly belongs to `offset`. It is common to choose the type based on the first parameter; in this example, 1, 2, and 4 would belong to `offset`, and 3, 5, and 6 would belong to `pointer`. However, except for 1, none of the operators can be implemented without access to the representation of the other type. There is no assignment of operators to types that makes this possible. The new types `pointer` and `offset` interact not only with each other: in order to have operations such as array subscripting, multiplication of offsets by integers is necessary:

```
7.  * :  offset  × integer → offset
8.  * :  integer × offset  → offset
9.  / :  offset  × offset  → integer .
```

As we define more operators it becomes increasingly difficult to classify them according to type: *all* types used in a program are related by the operations on them.

2.2. Polymorphic Examples

Polymorphism in a class-oriented type system is achieved using one of three devices: class hierarchies (as in Simula 67 and Smalltalk [Gol 83]), generic classes (as in Clu [Lis 77]), and class parameters (as in Russell [Boe 86, Don 85]). We describe a number of problems and how they may be addressed using these techniques.

From the previous section, recall that the new operator ** was defined for `integers` in terms of the existing operation *. The operation

```
** : t × integer → t
```

can in fact be defined for any type `t` provided that the operation

```
* :  t × t → t
```

exists. Using a class hierarchy, this problem is addressed by defining ** to belong to a super class, say `powerable_objects`. The class `powerable_objects` would contain a *virtual function* * requiring that all subclasses derived from it define the operation *. Then, the operation ** could be applied to any subclass of `powerable_objects`. In order to have ** apply to `integers` and `reals` it would be necessary to define the classes `integer` and `real` both to be subclasses of `powerable_objects`.

In some notations, concatenation of `strings` is denoted as multiplication:

```
* :  string × string → string
```

but such a definition is unlikely to be built into the class `string`. In order to define * and ** for strings, we would have either to redefine `string` to include * and make it a subclass of `powerable_objects`, or to create a new class `powerable_string` that has the the operations of both `strings` and `powerable_objects`. Such a definition requires that `powerable_string` be a subclass of *both* `string` and `powerable_objects`, which is impossible in a class hierarchy. In either case, making a small addition to the set of available operators forces a major restructuring of the class hierarchy.

In Clu, the class hierarchy has been abandoned in favour of the generic class. A generic class specifies a family of classes, and before being used a particular member of the family must be *instantiated*. A generic resembles a superclass, but rather than deriving subclasses we supply parameters to the generic to create an instance. For example, we could create the generic class `powerable`:

```
powerable :  class → class
```

Instances of the generic class might be `powerable[integer]` or `powerable[real]`. The definition of `powerable` would include the operator `**`; hence `**` is polymorphic. In order to define `**` it is necessary to constrain the possible type that may be used to instantiate `powerable`: In Clu, the programmer specifies the constraint that the parameter must *have* the operation `*`. Thus, we could not define `powerable[string]` unless `string` were first modified to contain `*`. In Russell, classes are values and there is therefore no distinction between generic and ordinary parameterization. One could therefore define an operator

```
** :  class × t × integer → t
```

where `t` is the value of the first parameter to `**`. The same structural constraint applies as with Clu: the class that `t` is bound to must contain the operation `*`.

We can extend our example of the types `pointer` and `offset` to make use of generic classes. The type `pointer` can be parameterized by the length of object to which the pointer refers:

```
pointer :  integer → class
```

Whenever an offset is added to such a pointer, its value is scaled by this amount. Thus, we wish to define the operations

```
+ :  pointer[n] × offset → pointer[n]
+ :  offset × pointer[n] → pointer[n]
```

for all `n`. In order to do operations such as field selection in records it is also necessary to define arithmetic between `pointers` that refer to memory blocks of differing lengths. Given a pointer to a record of length `n` we wish to index this pointer with an offset to yield a pointer to a field of length `m`. To this end, we define a class

```
field_offset :  integer × integer → class
```

which is parameterized by two integer parameters, `record_length` and `field_length`. We wish to define the operation

```
+ :  pointer[n] × field_offset[n,m] → pointer[m]
```

for all `n` and `m`. We know of no programming language other than our own that can express these types and operations.

3. Types in ForceOne

In **ForceOne** types and operators are autonomous entities; in any given scope a set (or *space*) of these entities is visible. Nesting and modules control the contents of a space; the type system controls the interaction among entities within the space. Here we present only the type system; the modularity and scope rules are unexceptional.

3.1. Basic types and declarations

The primitive types in **ForceOne** are `int`, `real`, `bool`, `char`, and `void`. Each has a pre-defined representation and a unique type mark. From the primitive types, new types are constructed using built-in and user-defined type generators. The built-in type generators are `ref`, `proc`, `record`, and `[ ... ]` (parameterization). A type generator constructs a new type mark and a new representation from the type marks and representations of its parameters. A given type generator will always yield the same type mark if its parameters are the same. Simple examples of generated types are `ref int`, `proc bool`, `ref proc bool`, etc.. Function types are generated by prefixing a formal parameter list to a type. For example, the type specification

```
[ real, int ] char
```

defines the type

```
real × int → char
```

It is sometimes necessary to name the formal parameters, but these names do not alter the type e.g.:

```
[ x: real, y: int ] char
```

A new type is introduced by a type declaration:

identifier : `type` == *type*

A type declaration always creates a unique type mark. Two functions are also defined by a type declaration:

`detype` : `[` *identifier* `]` *type*

and

`retype` : `[` *type* `]` *identifier*

These functions allow the programmer to convert the type of a value to and from that of its underlying representation, and are used in defining primitive operations on the new type. By restricting the visibility of `detype` and `retype`, the new type can be made *opaque*.

Ordinary values (including functions) are declared thus:

identifier : *type* == *expression*

Here, *identifier* is a new name bound to the value yielded by *expression*. If *identifier* is a function with a named parameter list, *expression* describes the function in terms of the names of the parameters, e.g.:

```
double: [ i: int ] int == 2 * i
```

If the parameters are unnamed, *expression* must yield a function of the appropriate type, e.g.:

```
fred: [ int ] int == double
```

3.2. Type generators

Type generators are functions that build new types. For example, the built-in generator `ref` is a function

```
ref : type → type
```

The programmer may define a type generator via a parameterized type declaration, e.g.:

```
record_offset :  [ int , int ] type == int
```

defines

```
record_offset :  int × int → type
```

An example of a type created from this generator might be

```
record_offset[12,4]
```

or

```
ref record_offset[36,1]
```

3.3. Overloading

Overloading – the ability to have many definitions for the same identifier – is fundamental to the type system. An identifier is overloaded only if each of its definitions is prefixed by the keyword `overload`, e.g.:

```
overload + :  [ a: int, b: int ] int == a - -b
```

Only identifiers so declared are overloaded; an `overload` declaration hides any previous non-`overload` declaration of the same identifier and a non-`overload` declaration hides all previous declarations.

When an overloaded identifier is used, the appropriate definition is chosen so as to match its type to that dictated by context. If there is no match, or if several match equally, an error results. This selection process is like the one for Ada [Ada 83], but is slightly more general in that identifiers of any type may be overloaded, functions may yield functions as results, and automatic conversions are allowed (automatic conversions are not essential to the type system and are not discussed further).

3.4. Implicit parameters

An *implicit parameter* is a parameter to a function that is not specified explicitly when the function is applied. Rather, the corresponding argument is selected by name at the application site. Without implicit parameters, a general function to take logarithms might be defined:

```
log :  [ a: real, log_base: real ] real
```

Using this definition, an engineer might apply `log[x,10.0]`; a mathematician might specify `log[x,e]`; and a computer scientist might use `log[x,2.0]`. `log_base` is made an implicit parameter by separating it from the rest by a `|`:

```
log:  [ a: real | log_base: real ] real
```

All applications of `log[`*expression*`]` are equivalent to `log[`*expression*`,log_base]`, and so the user can define the base to which logarithms are taken (within his environment) by declaring, for example

```
log_base: real == 2.0
```

A particular definition of `log_base` affects applications of `log` only within its static scope (that is, a computer scientist with `log_base = 2` could correctly call a function in a mathematical library in which `log_base = e`).

Implicit parameters are fully typed statically and are implemented exactly as explicit parameters. They give a function the ability to inspect the space of names and values available at the application site, and are a necessary facet of the polymorphic facilities in **ForceOne**.

3.5. Query parameters

In a function definition, it is not necessary to specify completely the type of a formal parameter. Instead, we give a *template* consisting of types, type generators, and query parameters. A query parameter has the form

`?` *identifier*

and appears in place of a type or in place of a parameter to a type generator: a query parameter is a wild card that matches anything. *identifier* is bound to the actual type or value that is matched by the wild card. For example, we may define a function that takes a parameter of any type and returns the same type:

```
echo: [x: ?t] t == x
```

`echo` as defined here is necessarily very simple: `t` is treated as a new type within `echo`, and hence has no operations (not even `detype` and `retype`). Any operations must be defined locally or passed as parameters; typically these parameters are implicit. For example, we may define exponentiation: †

```
**: [x: ?t, i: int | *: [t,t] t] t ==
            if i = 1 then x else x ** (i-1) * x
```

Typical applications of `**` might be:

† This definition of `**` works only for exponents greater than zero. To handle zero exponents, it is necessary that a value `multiplicative_identity` be defined. For negative exponenents, a `multiplicative_inverse` function is also required. These values could be specified as additional implicit parameters.

```
2 ** 3              -- 8
1.5 ** 2            -- 2.25
"abc" ** 3          -- illegal; no  *: [string,string] string
```

The third application could be made possible by defining a new *:

```
*: [a: string, b: string] string == concat[a,b]

"abc" ** 3      -- "abcabcabc"
```

Using query parameters, we can define the types pointer, offset, and field_offset, and the operations among them:

```
offset: type == int
pointer: [ i: int ] type == int
field_offset: [ i: int, j: int ] type == int

overload +: [o: offset, oo: offset] offset ==
                    retype[detype[o] + detype[oo]]
overload +: [o: offset, p: pointer[?n]] pointer[n] ==
                    retype[detype[o]*n + detype[p]]
overload +: [p: pointer[?n], o: offset] pointer[n] ==
                    o + p
overload +: [p: pointer[?n], f: field_offset[n,?m]] pointer[m] ==
                    retype[detype[p] + detype[f]]

overload -: [o: offset, oo: offset] offset ==
                    retype[detype[o] - detype[oo]]
overload -: [p: pointer[?n], o: offset] pointer[n] ==
                    retype[detype[p] - n*detype[o]]
overload -: [p: pointer[?n], pp: pointer[n]] offset ==
                    retype[(detype[p]-detype[pp]) / n]

overload *: [o: offset, i: int] offset ==
                    retype[detype[o] * i]
overload *: [i: int, o: offset] offset ==
                    o * i

overload /: [o: offset, oo: offset] int ==
                    detype[o] / detype[oo]
```

3.6. Nested query parameters

Nested formal parameters occur in defining functions that take other functions as arguments. In this situation, query parameters may be used in two senses that are best introduced by an example. Given the three functions

```
fred: [ i: int ] int ==  ...
john: [ j: real ] real == ...
dick: [ k: ?t ] t == ...
```

we wish to define the function takes_fred_or_john that will accept as a parameter a unary function on any specific type, like fred or john. We wish also to define takes_dick that will accept as a parameter only a polymorphic function like dick. The first is defined using an ordinary query parameter:

```
takes_fred_or_john: [ z: [?q] q ] int == ...
```

but the second is defined using a nested query parameter:

```
    takes_dick: [ w: [??r] r ] int == ...
```

The difference is that q is a parameter to takes_fred_or_john and is therefore bound to a particular value for its entire scope; r specifies a parameter to w, which is in turn a parameter to takes_dick: w does not take on a particular value within takes_dick (in fact its visibility is confined to the type specification of w).

4. Discussion

Our type system is amenable to solving not only these problems: from the primitives it is possible to build structures that are analogous to classes and class hierarchies, mutually dependent classes, and generic classes. In addition, without resorting to separate generic instantiation, we achieve very flexible polymorphism within a statically typed language. A compiler for **ForceOne**† has been implemented: all binding and type checking is done statically and all functions generate straightforward reusable object code.

We do not wish to claim that all class-oriented languages suffer from all the shortcomings described, but all of which we are aware have some. Those that fare best have done so by partially abandoning the notion that types are classes. Ada is perhaps the best example: by (partially) separating the notions of package and type, Ada handles the non-polymorphic cases described. However, to provide polymorphism, Ada reverts to generic modules, which re-couple the notions of type, parameterization, and packaging. Other languages that circumvent the class structure include C++ [Str 86], in which the *friend* facility allows a class to expose its representation to a procedure not defined in the class. A number of language designers have proposed the notion of *multiple inheritance* [Myl 80] in an attempt to make a class hierarchy fit the desired program structure. Highly polymorphic languages like Russell, Poly [Mat 85] and Smalltalk seem to adhere rigidly to the notion that a type *is* a class, and our structural criticism applies. Smalltalk also sacrifices static type checking in providing polymorphism.

In conclusion, we believe wholeheartedly that the purpose of types in programming languages is to *control* the operations that may be applied to particular values. However, the type need not *contain* these operations: we advance the type structure described here as being more fundamental.

Acknowledgements

The work of the first and second authors was supported by the Natural Sciences and Engineering Research Council of Canada. The work of the first and third authors was supported by **COGNOS** inc..

References

[Ada 83] *Reference Manual for the Programming Language Ada,* U.S. Department of Defense, ANSI/MIL-STD-1815-A (1983)

[Boe 86] Boehm, H. and Demers, A., *Implementing Russell* A.C.M. Sigplan Not. 21:7 (1986), 186-195.

[Cor 86] Cormack, G.V., and Wright, A.K., *Polymorphism in a Compiled Language*, Univ. Waterloo CS-86-27 (1986)

[Dah 70] Dahl, O.J., Myhrhaug, B. and Nygaard, K., *The Simula 67 Common Base Language, Norwegian Computing Centre S-22 (1970)*

[Don 85] Donahue, J. and Demers, A., *Data Types are Values,* A.C.M. Trans. Prog. Lang. Syst. 7:3 (1985), 426-445.

[Gol 83] Goldberg, A. and Robson, D., *Smalltalk-80, the Language and its Implementation*, Addison-Wesley (1983)

[Lis 77] Liskov, B.H., Snyder, A., Atkinson, R. and Schaffert, C., *Abstraction Mechanisms in Clu,* Commun. A.C.M. 20:8 (1977), 564-576.

† We compile implicit parameters, query parameters, and built-in type generators; user-defined type generators are not yet included, but their implementation does not differ in essence from that for built-in type generators.

[Mat 85] Matthews, D., *Poly and Standard ML,* A.C.M. Sigplan Not. 20:9 (1985), 52-76.

[Myl 80] Mylopoulos, J., Bernstein P.A. and Wong, K.T., *A Language Facility for Designing Database-Intensive Applications,* A.C.M. Trans. Prog. Lang. Syst. 5:2 (1980), 185-207.

[Str 86] Stroustrup, B., *The C++ Programming Language,* Addison Wesley (1986)

[Wri 86] Wright, A.K., *Reference manual for the language* **ForceOne**, unpublished (1986)