A LOGICAL VIEW OF TYPES

Patrick J. Hayes and Gary G. Hendrix
SRI International

I INTRODUCTION

People working in the areas of data abstraction, databases, and conceptual modeling have argued at considerable length over the meanings of such terms as "abstraction" and "type."* Rather than add to this debate by offering yet another set of definitions, in the paragraphs below we shall attempt to show how ordinary predicate calculus can be used to talk about most (perhaps all) the notions for which the terms "abstraction" and "type" are currently being used in various quarters of computer science. We do not intend to argue that predicate calculus is a suitable tool for implementing types, but rather that it provides a well-understood, uniform conceptual framework and notation for describing and precisely comparing various ideas on typing--and that special notations developed for this purpose are therefore unnecessary.

II THE BASIC NOTION

We may argue at length over what kind of thing a type should be taken to be, but most of us can agree that it is generally meaningful to say that an object is or is not of a certain type. That is, statements of the form

"b is of type t"

make sense, at least in English.

Perhaps the most neutral approach to types is to say that being of a certain type can be taken to define a property of objects.** Now, being of a certain type is an interesting property only insofar as one can infer other things from this fact; in other words, only insofar as there is a useful theory of the properties of "being of a type." For example, one might know such things as

(1) All objects of type t have property p.
(2) Objects of type t are never objects of type s.
(3) Objects of type s are also of type u.
(4) All objects of type u have property q.

These would be part of the theory of what it is to be of type t. From these and the single fact that b is of type t, we can infer that

(1a) b has property p.
(2a) b is not of type s.
(3a) b is (also) of type u.
(4a) b has property q.

In essence, the utility of talking about some object's being of a given type is that "being of the type" entails a number of other consequences. For example, statements such as 1-4 provide general information about what "being of the type t" entails. By associating the general information with the type (whatever a type is), the information need not be explicitly recorded for every particular object that is "of the type." The result is a saving of storage and an ability to think collectively about groups of objects that share properties in common.

III NOTATION

Without making any commitment as to what type t is, or what it means to be of type t, we can adopt the notation "T(b)" to indicate that "b is of type t." That is, we can represent "being of type t" by a unary predicate T.

Using this notation to denote type information, the English statements 1-4 may be stated formally as

(1b) T(x) => P(x)
(2b) T(x) => NOT S(x)
(3b) T(x) => U(x)
(4b) U(x) => Q(x).

All of these have a rather special form. Each contains only unary predicates. This seems to be a general feature of typing theories; they are all expressed in the unary fragment of predicate logic. One might hypothesise, as has SRI's Robert C. Moore, that the function of types is to provide efficient machinery for handling this particular decidable and eminently utilitarian subtheory.

* This research was supported by the Defense Advance Research Projects Agency with the Naval Electronic Systems Command under contract N00039-79-C-0118. The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Defense Advance Research Projects Agency of the United States Government.

** According to some views of what types are, one cannot identify a type t with the property of having type t: nevertheless that property is always well-defined.

Moreover, the above facts are all universally quantified and, with but one exception, are simple implications between atoms. These implications seem to make up the bulk of most systems based on type; they are the facts that express the "inheritance" relationships among types. In the next section we shall consider relaxing the atomic constraints on the form of a type theory, but the restriction to unary predicates will be retained throughout.

IV SPECIAL CASES

In the literature, various types (pardon the pun) of typing have been singled out for special attention. These all are expressible in the present framework by defining special forms of unary predicate.

A. Specialization by Adding "Slots"

For example, one might say that t is a specialization of some supertype s if t adds to s some new "slot" or "role" besides those required to be of type s. For example, the event type traveling-in-a-vehicle is a specialization of the event type traveling, since the latter would allow traveling by foot. Traveling by vehicle requires a new "slot"--the vehicle.

If being a traveling is expressed by S, then we can define the slot-enriched type property T by the definition

(5) $T(x) = S(x)$ AND (EXISTS y)(y=f[x] & U(y)),

from which it follows that

(6) $T(x) \Rightarrow S(x)$,

so that T inherits attributes from S, as required. Formula (5) says that if you can find an x of type t, there will then exist a y (the "slot filler") that is associated with x. This y is functionally related to x by f (f is the name of the "slot" or "role"), and y is of type u (which restricts y's possible values). In our example, u is the type for vehicles.

B. Specialization by "Restricting Slots"

One might also say that t is a specialization of supertype s if t adds to s the new restriction that the filler of some "slot" f has some restricting property R. (R might even restrict the value to a constant.) If

$S(x) \Rightarrow$ (EXIST y)(y=f[x] & U(y)),

then the property of having the restricted type t is defined by

$T(x) = S(x)$ AND R(f[x]),

where R(y) places restrictions on y not imposed by U(y). It follows from this definition that

$T(x) \Rightarrow$ (EXIST y)(y=f[x] & R(y)).

V FORM AND CONTENT

It is perhaps worth emphasising that nothing in the foregoing text makes any assumption concerning the meaning of the type predicates. The objects being classified can be anything: numbers, physical objects, events, times, possible worlds, ships, people, and so forth. In particular, a given problem domain can often be split up into individual objects and these individuals classified into types in a variety of ways--some of which may be better for certain purposes than others.

As an example, let us consider the problem of how to describe events and changes in time. One approach, now so widely used that it is almost classical, is to introduce objects called states or situations, forming a distinct type of their own, that represent instantaneous "snapshots" of the universe. These things give rise to a number of problems, including the well-known frame problem. But the same actual world can be carved up into individual pieces differently; this is done by introducing objects called processes or histories, also forming a distinct type, intended to be separable pieces of the four-dimensional universe that extend in both space and time and have natural boundaries.

The same language and machinery of types can be used in either case, and the same external world is being described, but the descriptions differ markedly from one another--not least in the type axioms that arise in each case. It is often mistakenly assumed that the problems stemming from state-based descriptions are problems of the logical language. It is a merit of the use of logic for expressing type information that this sort of confusion between questions of form and those of content is easily diagnosed.

VI DATA TYPES

In attempting to incorporate the notion of types into programming languages, nasty problems can arise from ignoring the distinction between symbols and the objects they are to represent. The root of the difficulty is that computer systems often represent objects by symbols that have internal structures of their own. For example, one might talk about the point P, using the letter "P" as an unstructured symbol to designate the point; but a computer system might designate P by using a symbol that has at least three structural properties: a bit sequence of length n representing the abscissa, a bit sequence representing the ordinate, and a location in memory.

In specifying type information in a computer language, all too often it is unclear whether one is making a statement about things, about the representing symbols, or about both. What are we to make of statements such as "A point is a pair of reals?"

This confusion arises naturally from a tradition in programming-language design that began

with the early autocodes, which were thought of as high-level abbreviations of assembly code. In assembler language there are several representations available for certain common data structures, notably numbers (integers and reals). The autocoder, however, did not want to be bothered by these distinctions, so types were introduced to denote the distinctions needed by the compiler rather than the programmer. This notion of type refers to the inner structure of the [data structure that implements the] symbols being manipulated by the language system, not the logical relationships that hold among the things the symbols represent.

The confusion is compounded by the recent trend in language design, started in SIMULA and continuing in SMALLTALK, APIARY, CLU, ALPHARD, etc., to treat the run-time structures of the system as objects themselves, to conflate these objects and the data structures of more traditional languges, and to think of the program text as being attached to these things as well as describing them. This gives rise to an intuitive picture in which the objects the language describes--the things that are of a type--are little active agents, each with its own script, running around sending messages to one another, etc. While this picture does have some intuitive appeal, it is hard to reconcile with any coherent relationship between the language and some external world.

But this confusion is surely not necessary. Even though these internal "objects" have a structure that arranges them in some kind of type hierarchy, they are essentially syntactic entities, and this "internal" type hierarchy follows from their internal syntactic structure as symbols. For interesting cases there will also be a real type structure that reflects their role as descriptors of some external world.

To avoid confusion, one needs to keep these two types of type quite distinct, albeit united in a common framework. One needs theories of the structure of symbols and of the manner in which symbols are being used to represent objects. Such theories will involve formulas such as

```
(ALL x)(EXISTS y) S(x) & O(y) => DESIGNATES(x y),
```

which should be interpreted as "if x is any symbol of type S, then there is an object of type O which x designates."

Similarly, we might have

```
(ALL x, y) S(x) & O(y) & DESIGNATES(x y)
              => DESIGNATES(f[x]  g[y]).
```

which says that, if symbol x of type S designates object y of type O, then the "filler" of some "slot" f in x designates the value of some attribute g of y.

We do not have a complete theory of this kind, and to construct one is certainly a major research project. Nevertheless, we do claim that predicate logic is the only well-understood framework in which such a theory could forseeably be developed, not least because it does not beg any of the important questions such a theory must answer.

VII CONCLUSION

We began by making minimal assumptions about types. But the simple idea of the property defined by a type seems not only to allow us to say anything we would wish to say about types, but also averts several major sources of confusion. So why bother with anything more? And using predicate logic, all or this can be expressed elegantly, clearly and with a refreshing minimum of fuss.