A value can belong to many types

B H Venter
Department of Computer Science, University of Port Elizabeth
P.O. Box 1600, Port Elizabeth, 6000 South Africa
e-mail: csabhv@upe.ac.za

Abstract

Most statically typed Algol-style languages were designed to conform with the view that a value belongs to one and only one type. The adoption of this view bought compiler simplicity at the expense of language expressivity. This trade off has been partially offset by the introduction of subtypes. The adoption of the view that types may intersect in arbitrary ways, on the other hand, requires complicated compilers or costly run-time type checks. However, language expressivity is enhanced, language semantics are simplified and object-oriented and database concepts can be integrated smoothly with the Algol style.
Keywords: *Data abstraction, object-oriented programming, polymorphism, type checking*
Computing Reviews Categories: *D.3.3*

1 Introduction

In his influential early paper on data types[4], Hoare states:

> Every value belongs to one and only one type.

It is evident from Algol-style languages like Ada and Modula that many programming language designs conform with this view.

The adoption of this view bought compiler simplicity at the expense of language expressivity — a reasonable trade off then, but not now. In recent languages, such as Oberon, some of the lost expressivity is regained by the introduction of subtypes[13].

It is shown in section 10 that, in real life, types intersect in ways that are difficult to model using only sub typing. A closer fit between reality and model can be achieved if types are allowed to intersect in arbitrary ways. The aim of the paper is to examine some of the consequences and to suggest how arbitrary intersections can be constructed.

2 Types as sets

If one views a type as merely a set of values, type intersection is straightforward to define and understand (it is merely set intersection). If one adopts the classical view that a type is an algebra[8], however, it is much more difficult to define the meaning of intersection.

The author contends that it is not essential to view a type as an algebra. Types are useful because they provide structure and allow type checking. This can be done even if one views types merely as sets of values. In the rest of this paper, therefore, types are sets of values and $\mathcal{U}$ will denote the set of all the values that can belong to a type.

The author contends further that in programming languages types play only one role: they serve as constraints on the domains and codomains of functions. That is, if $\mathcal{D}$ and $\mathcal{C}$ are types, and $f : \mathcal{U} \rightarrow \mathcal{U}$ is a function with domain constraint $\mathcal{D}$ and codomain constraint $\mathcal{C}$, then all programming languages require the definition of f to conform to the following:

$$\begin{array}{ll} f(x) = \textbf{Undefined} & \text{if } x \notin \mathcal{D} \\ f(x) \in \mathcal{C} \cup \{\textbf{Undefined}\} & \text{if } x \in \mathcal{D} \end{array}$$

Typically, an error is signaled when $f(x)$ returns **Undefined** (not be confused with non-termination or Bottom). In statically typed languages such errors are sometimes predictable by the compiler.

As a syntactic convenience, we will write $f : \mathcal{D} \rightarrow \mathcal{C}$ to show that $f : \mathcal{U} \rightarrow \mathcal{U}$ is a function with domain constraint $\mathcal{D}$ and codomain constraint $\mathcal{C}$.

The reader may be forgiven for being sceptical about the contention that types only serve to constrain functions. After all, types are used to declare variables and check assignments. What's that got to do with functions?

Everything, as long as we regard variables as functions whose definitions are provided and updated by assignment statements. In other words, a variable declaration such as

V : Some-type

must be regarded as syntactic sugar for

> v : $\{\} \rightarrow$ *Some-type* is a function whose evaluation rule will be supplied later on by an assignment statement and may be changed (as time progresses) by further assignment statements.

We will furthermore regard V to stand for $v()$ when used in an expression (v can be denoted as Address(V)).

3 Related work

Value sharing (type intersection) has received scant attention from researchers working on type systems. For example, Cardelli[2] mentions it in passing as a special case of parametric polymorphism but gives no explicit consideration to it. Language designers either dispense altogether with the notion of typed variables (for example Icon[5] and SETL[9]) or make every effort to restrict value sharing.

As far as the author knows, the work of Bailes on G[1] has been the sole exception to this trend. Bailes, however, arrives at value sharing as a necessary consequence of his well-motivated overall approach and gives little specific attention to it.

4 Type equivalence

The question of when type A is equivalent to type B has been difficult to resolve using other views of what a type is. If, however, types are viewed as sets that merely serve as domain and codomain constraints on functions, there is only one sensible answer: type A is equivalent to type B, *if and only if* the same set of values make up both type A and type B.

Thus, if we declare

type A = **array** 1..10 **of** Int
type B = **array** 0..9 **of** Int

we can determine whether types A and B are equivalent merely by determining whether *set* A is equal to *set* B.

Of course, the difficult question now becomes the interpretation to be given to type constructors. Most type constructors have straightforward interpretations, as can be seen from the following examples:

set of T $= \{S \mid S \subseteq T\}$
array R **of** T $= \{f \mid f : R \rightarrow T\}$
pointer to T $= \{f \mid f : \{\} \rightarrow T\}$

We thus can readily conclude that types A and B above are not equivalent since $f \neq g$ for all $f : \{1 \ldots 10\} \rightarrow \text{Int}$ and $g : \{0 \ldots 9\} \rightarrow \text{Int}$. But what about the following types?

type C = **record** x : Int; y : Char **end**
type D = **record** x : Int; y : Char **end**

The question should be familiar to participants in the name equivalence versus structural equivalence debate. By interpreting types as sets, the question is transformed, however, from "Are types C and D equivalent?" into "How does one characterize the set of values represented by a record constructor?". Neither question has an answer that will satisfy everyone.

Given our convention of interpreting a variable name, such as R, as syntactic sugar for $r()$, it seems reasonable to regard R.x as syntactic sugar for $r()(x_C)$ and S.x as syntactic sugar for $s()(x_D)$, if R is of type C and S of type D. This requires the following interpretation of type C as a set:

$\{f \mid f : \{x_C, y_C\} \rightarrow \text{Int} \cup \text{Char}$
  and $f(x_C) \in \text{Int}$ **and** $f(y_C) \in \text{Char}\}$

Thus, if $x_C = x_D$ and $y_C = y_D$ then types C and D are equivalent. We now have two sensible choices:

1. $x_C = x_D =$ 'x' and $y_C = y_D =$ 'y'.

2. x_C, y_C, x_D and y_D are distinct values.

Selecting the first choice amounts to a form of structural equivalence that allows the conclusion that type E, below, is also equivalent to types C and D.

type E = **record** y : Char; x : Int **end**

Selecting the second choice amounts to name equivalence.

5 Assignment compatibility

By our earlier definition, a variable is a function and the value returned by this function must be a member of the set $\mathcal{T} \cup \{\textbf{Undefined}\}$ where $\mathcal{T}$ is the type of the variable. When an expression is assigned to a variable, it is thus necessary to ensure that the expression results in a member of $\mathcal{T} \cup \{\textbf{Undefined}\}$ before allowing the assignment to take effect[1].

Since operators are also functions, expressions can be regarded as function applications. The type of an expression is thus the range set of the (outer) function that is being applied. This is guaranteed to be a subset of $\mathcal{C} \cup \{\textbf{Undefined}\}$ where $\mathcal{C}$ is the codomain of the function.

[1]Usually, assigning **Undefined** to a variable will also result in a run-time error.

Now, if $\mathcal{C} \subseteq \mathcal{T}$ or $\mathcal{C} \cap \mathcal{T} = \{\}$, the validity of the assignment can be decided without evaluating the expression. Thus, a language designer can ensure that all type checks can be done by a compiler, simply by insisting that assignments are illegal unless $\mathcal{C} \subseteq \mathcal{T}$.

Compile-time (or static) type checking is generally regarded as highly desirable and hence many language designers claim that their languages allow static type checking. In practice, however, static type checking is so restrictive that no realistic language enforces it. Virtually all languages allow assignments to be carried out if $\mathcal{C} \supseteq \mathcal{T}$ and generate run-time checks to ensure that the assignments are legal. The main difference from language to language is whether the language forces the programmer to make the type check explicit or not.

If types are to intersect *freely*, however, it is not reasonable to insist that $\mathcal{C} \supseteq \mathcal{T}$. Assignments should be allowed, subject to run-time checks if necessary, whenever $\mathcal{C} \cap \mathcal{T} \neq \{\}$.[2]

Of course, this means that many type errors that are easily detected by the compilers of languages like Ada and Oberon, become very difficult for a compiler to detect, thus requiring the generation of expensive run-time checks. The author contends, however, that it is no longer appropriate to restrict the expressivity of a language merely to limit the complexity of compilers and run-time systems. *Such a strategy does not get rid of the complexity*, it merely dumps it over the fence in the application programmer's yard. Furthermore, much progress is being made on the optimization of "dynamic" languages (for example [3]) and it seems reasonable to expect that such compilers will convert most run-time errors into compile-time errors.

That is, while static languages may be safer than dynamic languages at present, they are not *inherently* safer.

6 Problems with records

Although the second interpretation for record constructors (amounting to name-equivalence) has been dubbed sensible in section 4, it does *not* seem sensible to adopt this convention in a language where types are regarded merely as sets, since the obvious interpretations of the sets of values generated by the other type constructors all amount to structural equivalence.

Structural equivalence, however, creates problems when records are used as the "concrete" representations of abstract types. For example, if

```
type Stack =
  record
    Top : range 0..100;
    Elems : array 1..100 of Int;
  end record
```

and

```
type Funny-rec =
  record
    Top : range 0..100;
    Elems : array 1..100 of Int;
  end record
```

then types Stack and Funny-rec represent the *same* set of values if structural equivalence is used.

[2] This is already the case for subrange types in most languages. The argument here is that it should apply to other types as well.

Given assignment compatibility as outlined in the previous section, this means that a client programmer can always make the "hidden" representation of abstract types visible by providing alternative type definitions that happen to describe the same sets of values as the abstract types. Thus, abstract types have to be handled in non-traditional ways if types are to intersect freely.

7 Types as values

One intuitively expects to be able to declare variables of type "**set of** something". If one allows such types, it implies that one allows sets to be elements of sets, which implies that one ought to allow types to be values[3].

Of course, if a programming language allows very large sets, like Int, to be regarded as values, the language implementations must resort to a form of lazy evaluation: one would not wish the compiler to generate code that fully evaluates a set like $\{x \in \mathrm{Int} \,|\, \mathrm{odd}(x)\}$ if the set is only used as the target of a membership test.

8 Dynamic Types

Most programming languages require all the values making up a type to be determinable during compilation, since this is a prerequisite for static type checking. If one drops the insistence that type checks should be static where possible and very simple otherwise, it becomes possible to consider whether sets of values, *that vary at run-time*, should be allowed to serve as types.

Such "dynamic" types would be very useful in database-oriented languages. For example, one would be able to declare:

Parts : **set of** $\mathcal{U}$
Suppliers : **set of** $\mathcal{U}$
Parts-supplied : Suppliers $\rightarrow$ **set of** Parts

Since Suppliers is a *variable*, values of *type* Suppliers can be created and destroyed as time progresses, simply by including and excluding values from *set* Suppliers. Moreover, the domain constraint of function Parts-supplied dynamically adjusts to the creation and destruction of supplier values. (In other words, the type system takes care of referential integrity.)

Although *any* sets of values can make up types Parts and Suppliers, it is better not to press numeric, string, array or record values into service as part or supplier values. Parts and Suppliers are abstract data types and a means must be provided to populate them with abstract values. In other words, it must be made possible to get hold of values that belong to $\mathcal{U}$, but to no other set. Providing a built-in function called *new* comes to mind.

By making it possible to create new values at run-time and to include these values into one or more run-time sets that can serve as *types*, one makes it possible for types to intersect in arbitrary ways. With classical sub typing, the set of types to which a newly created value can belong can be determined by the compiler and types can only intersect if the one is a subtype of the other or if both types share a common subtype.

[3] If $\mathcal{U}$ is treated as a value, it becomes possible to specify paradoxical sets, for example $\{\mathcal{S} \subset \mathcal{U} \mid \mathcal{S} \notin \mathcal{S}\}$. To avoid paradoxes we decree that $\mathcal{U} \in \mathcal{U}$ and $\mathcal{U} \subset \mathcal{U}$ by definition and that all paradoxical set formers must result in **Undefined**.

9 Object Classes

Dynamic typing subsumes sub typing and provide for the integration of object-oriented concepts and Algol-style languages in a natural way. For example

$$\text{Suppliers} : \textbf{set of } \mathcal{U}$$

can be seen as the declaration of an object class and

$$\text{Parts-supplied} : \text{Suppliers} \rightarrow \textbf{set of } \text{Parts}$$

can be seen as the declaration of a member variable for class Suppliers. Dynamically creating new abstract values and including them in *set* Suppliers amounts to dynamically creating objects of *class* Suppliers.

Derived classes correspond to subtypes, and inheritance follows naturally. For example, given type (class) $\mathcal{A}$, an operator (method) $f : \mathcal{A} \times \mathcal{A} \rightarrow \mathcal{A}$, and a subtype (derived class) $\mathcal{B} \subset \mathcal{A}$, then f may obviously be applied to arguments of type $\mathcal{B}$. Moreover, the value resulting from the application of f to arguments of type $\mathcal{B}$ may well be a member of $\mathcal{B}$ and thus may be assigned to a variable of type $\mathcal{B}$, subject to a "run-time" type check. In other words, subtype (derived class) $\mathcal{B}$ effectively inherits operator (method) f from supertype (super class) $\mathcal{A}$.

Inherited methods can be overridden by allowing operators (methods) to be overloaded. However, operator/function names cannot be as freely overloaded as in a language like Ada. For example, if

$$f : \mathcal{A} \rightarrow \mathcal{C},\ f : \mathcal{B} \rightarrow \mathcal{C}, \text{ and } \mathcal{A} \cap \mathcal{B} = \{\text{ab}\}$$

then it is impossible to determine which f must be used when one encounters $f(\text{ab})$.

Similar difficulties arise in object-oriented languages that allow multiple inheritance and similar solutions can be adopted.

10 Beyond Classes

Object-oriented programming languages are currently advocated by many as the long awaited solution to the software crises. For the most part, the author agrees that these high expectations are justified. However, it should not be imagined that these languages have advanced programming language expressivity to the natural limits.

For example, consider an object-oriented design for a university database. It seems likely that the designer will have introduced a class Person with sub classes Student and Staff-member. The usual interpretation of classes as types require Student and Staff-member to be disjoint. In real life, however, these sets are not disjoint and the designer thus will have to go further.

In simple cases, multiple inheritance seems the answer. If the designer adds a Student-staff-member class, which inherits from both Student and Staff-member, to the database schema the problem seems solved: An instance of Student-staff-member ISA Student and ISA Staff-member, the latter two classes are thus no longer disjoint.

Consider, however, the complications that arise when using this technique to model the real life intersections between classes Student, Staff-member, Parent and Sports-club-member. Must the database designer anticipate the need for the class Sporting-parent-student-staff-member? Surely not!

Consider further, the complications that arise if the designer decides to introduce sub classes Academic-staff-member, Administrative-staff-member, Technical-staff-member, and so on. In this case, creating an instance of Student-staff-member creates a staff member object that belongs to none of the sub classes of Staff-member.

The designer thus is forced to remove class Student-staff-member from the schema and replace it with Student-academic-staff-member, Student-administrative-staff-member, and so on.

The author suspects that the database designer will simply give up and provide a schema that is an oversimplification of the real world. The application programmer ends up having to solve the problem, typically by adding explicit run-time type checks all over his code.

If the designer can use dynamic types that can intersect in arbitrary ways, the above modeling problems simply do not arise.

11 Summary

The consequences of *completely* abandoning the view that every value belongs to one and only one type are profound. One can achieve no more than a *partial* abandonment, however, unless one also holds that a type is simply a set of values and allow *any* set of values to serve in the role of a type.

The interpretation of a type as no more than a set of values forces one to conclude that type constructors are simply set operators. This in turn simplifies the resolution of questions about type equivalence and suggests that structural equivalence is appropriate.

Structural equivalence leads to problems when applied to record constructors. Records have also proved troublesome to database designers[6] and have been successfully done away with in DAPLEX[10]. This suggests that records are not "nice" constructs and that "post-Hoare" programming languages might better off without them.

Another consequence of regarding types as nothing more than sets of values, is that operators take on an independent existence and that the notion of operator inheritance by subtypes thus becomes simple. Also, types can be made into first-class values by treating sets as first class values. This is more intuitive than other approaches[7], but requires implementers to tackle lazy evaluation. Finally, polymorphism and the ability to do type membership tests are automatic consequences of making types equivalent to sets.

In classical Algol-style languages the prevalence of disjoint types and highly restricted mechanisms for introducing subtypes make "nearly" static type checking possible. When any set of values can serve as a type, however, it must be accepted that non-trivial run-time type checks will have to be performed in some situations — typically those situations where the application programmer would have to provide them if the language does not.

Acceptance of non-trivial run-time type checking makes it reasonable to accept furthermore that, in some cases, even the values making up a type will not be known at compile-time. This leads to the very useful concept of a dynamic type that makes it possible to support abstract types in a non-traditional, more natural way, and to integrate concepts from database languages and object-oriented languages into Algol-style languages.

The author has recently designed an Algol-style language that supports and exploits unrestricted type intersection. A prototype version has been in use since early 1989 and our experiences with its expressivity are very positive. The design of the type system of the language is reported in more detail in [12] and the language itself is defined (in under 20 pages) in [11].

The author has little doubt that the active exploration of the design space of languages that allow and exploit unrestricted type intersection will result in some very exiting and powerful new programming languages appearing on the scene.

12 Conclusion

If one assumes that the "software crisis" will not allow restrictions on the expressivity of programming languages to survive advances in compiler technology, it is also safe to assume that statically typed languages like Ada, C++, Eiffel and so on have no future.

The author believes that the future belongs to persistent, polymorphic, dynamically typed, object-oriented languages that allow types to intersect freely. This article has shown that these need not be radical languages that are as different from Algol family as LISP, Prolog or Smalltalk are, but can just as well be Algol-style languages.

References

[1] P. A. Bailes. The hierachical development of a generic type mechanism for functional languages. *Computer Languages*, 15(1):1–26, 1990.

[2] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *ACM Comput. Surv.*, 17(4):471–522, Dec. 1985.

[3] C. Chambers and D. Ungar. Customization: Optimizing compiler technology for SELF, a dynamically-typed object-oriented programming language. *SIGPLAN Notices*, 24(7):146–160, July 1989.

[4] O. J. Dahl, C. A. R. Hoare, and E. W. Dijkstra. *Structured Programming.* Academic Press, 1972.

[5] R. E. Griswold and M. T. Griswold. *The Icon programming language.* Prentice-Hall, 1983.

[6] W. Kent. Limitations of record-based information models. *ACM Trans. Database Syst.*, 4(1):107–131, March 1979.

[7] J. C. Mitchell and G. D. Plotkin. Abstract types have existential type. *ACM Trans. Prog. Lang. Syst.*, 10(3):470–502, July 1988.

[8] J. Morris. Types are not sets. In *ACM Symposium on Principles of Programming Languages*, pages 120–124, New York, 1973. ACM.

[9] J. T. Schwartz, R. B. K. Dewar, E. Dubinsky, and E. Schonberg. *Programming with sets.* Springer-Verlag, 1986.

[10] D. Shipman. The functional data model and the data language DAPLEX. *ACM Trans. Database Syst.*, 6(1):149–173, March 1981.

[11] B. H. Venter. The programming language slim. Computer Science Technical Report CS-Venter89a, University of Fort Hare, Private Bag X1314, Alice, Ciskei, 1989.

[12] B. H. Venter. Data structuring via functions. *South African Computer Journal*, 1(2):45–50, May 1990.

[13] N. Wirth. Type extensions. *ACM Trans. Prog. Lang. Syst.*, 10(2):204–214, April 1988.