TYPE EQUIVALENCE IN STRONGLY TYPED LANGUAGES: ONE MORE LOOK

by

{Daniel M. Berry[1], Richard L. Schwartz[2]}

Faculty of Mathematics
Weizmann Institute of Science
Rehovot, Israel

Keywords: Language Design, Type Equivalence, Strong Typing, ALGOL 68, Euclid.

Introduction

Despite quite a few published papers examining the issue of type equivalence (e.g., [WSH 1977, Pop 1977, Ten 1978]), we see the need for further clarification of the issues surrounding type equivalence. We have recently witnessed several current research languages introduce breaches in their (strong) type systems because of an inappropriate definition of type equivalence. In this paper we re-examine the fundamental approaches to type equivalence, pointing out several problems in current proposals. It is suggested that in the absence of complete data type encapsulation, the Euclid [Lam 1977] approach provides a good way of introducing optimal type encapsulation without sacrificing data type security.

There appears to be general agreement that strong typing in programming languages is desirable. In a strongly typed language, each value has a unique type. The key impact of this is that, starting with a knowledge of the type of each identifier and constant appearing in the program, it is possible to determine the type of every expression in the program.

Given this complete typing of expressions in the language, an important issue to be resolved is the question of when two types are

---

[1]Also at Faculty of Mathematics, Hebrew University, Jerusalem, Israel. On sabbatical from Computer Science Department, UCLA, Los Angeles, California, 90024, U.S.A. Work supported in part by Department of Energy (USA), Contract EY-76-5-03-0034, PA 214, ONR Contract N00014-78-C-0656, and the Lady Davis Foundation (Israel).

[2]Work supported in part by ONR Contract N00014-78-C-0656, and a Weizmann Post-Doctoral Fellowship at the Weizmann Institute of Science (Israel).

equivalent. The answer to this question determines the usability of values of a given type. In general, where a value of a given type is usable, any value of an equivalent type may be used in its place. The resolution of type equivalence determines, for example, the compatibility of the source and target in assignments and during parameter passing. In type-extensible languages such as Pascal [JeW 1975], Algol 68 [vWi 1975] and almost all recent languages, this question becomes particularly critical*. With the possibility for the user to define new types in terms of existing ones, the equivalence of defined and defining types is non-trivial.

Structural Type Equivalence

The structural type equivalence approach, that used by Algol 68, says that all types with identical structure, i.e. implementation, are the same type. The type declaration

```
mode newmode = oldmode
```

introduces a new type newmode and causes it to be equivalent to oldmode. In the simplest case, newmode serves as an abbreviation for the possibly more complex type expression oldmode. This is the case for example in a type declaration such as

```
mode complex = struct(real re,im)
```

Because of the possibility of mutually recursive type definitions, newmode is not always simply an abbreviation for oldmode, and the question of type equivalence becomes complicated. For example, in Algol 68 the three types a, b, and c defined below are in fact all equivalent:

```
mode a = struct(int x,ref a y)
mode b = struct(int x,ref c y)
mode c = struct(int x,ref b y)
```

Loosely speaking, they are equivalent because when their recursive definitions are expanded into infinite trees with edges marked by selectors and leaves by basic types, they produce identical trees. This is very loosely speaking, as the formal statement of when two types are equivalent constitutes a major part of the W-grammar syntax for

*Unfortunately, the Pascal Report [JeW 1975] and axiomatic definition [HoW 1973] do not address this issue. This has led to differing decisions being taken by different Pascal implementations [Ten 1978] and inconsistent decisions within implementations [WSH 1977].

Algol 68 and is probably the most foreboding part of the syntax to the reader (Note that the fact that the rules are in the syntax indicates an intent for compile-time enforcement). Complaints about this complexity are a major reason for Algol 68's lack of popularity.

The problem with the Algol 68 approach is that it negates the protection that strong typing is to provide. Given the declarations

```
int i; bool b
```

One cannot do either of

```
i:=b
```

or

```
b:=i.
```

If, however, one has declared

```
mode newint = bool; ¢ integers in range 0 and 1¢
newint i; bool b
```

One is allowed to do both

```
i:=b
```

and

```
b:=i
```

simply because newint and bool are equivalent types. In addition, any operation defined for either type works on values of the other type.

One of the ramifications of this is that it is not possible to introduce true abstract data types. Each new type introduced via a type definition is simply another name for the defining type. Thus, there is no way to hide the implementation of an introduced data type.

Complete Encapsulation

The other extreme in the issue of type equivalence is to force complete encapsulation of defined data types. In complete encapsulation, a type is defined in some closed scope (e.g., module or cluster [LiZ 1974]) which hides the representing type. The module then contains the representation of the type and the bodies of all operations requiring knowledge of the representation of the type. The only identifiers visible outside the construct are the type identifier itself and the operation identifiers. The language (e.g., as proposed in [BEL 1977]) is constructed so that each definition makes a new type and only the operations defined within the type have access to the representation of the type. All other routines working on values of the type must be programmed using these visible operations. The language is also constrained so that the only way to introduce a new type is through such an encapsulated definition. Thus, all programming is done with encapsulated, abstract types (including the built-in basic types which are by nature encapsulated since their implementation is hidden). This may be best from the programming point of view because

it forces a greater degree of modularity and independence.

The drawback to complete encapsulation is that it may be too extreme in its dictation of program style in that it forces *all* type definitions to be encapsulated. An encapsulated definition is inconvenient (and inefficient) when the defined type is to inherit *all* the operations of the representing type and does not allow the introduction of a defined type as an equivalent abbreviation for the representing type. Because of these considerations, we address languages in which encapsulation is provided as an option but is not required. With such an approach, what to do with unencapsulated data type definitions remains.

Compromise Definitions of Type Equivalence

Some languages try to provide some kind of compromise between the Algol 68 structural equivalence and the complete encapsulation approach.

The simplest of these compromises, the so-called *name-equivalence* [WSH 1977] approach, is used in the U.S. Department of Defense Red language proposal [DOD 1978a]. In this proposal, each occurence of a type constructor, e.g. array, introduces a new type. Thus, to capture a constructed type for use in several declarations, the constructed type must be given a name in a type declaration, e.g. as in

```
type matrix = array 1..n,1..n of real  .
```

The language may provide that only named types, i.e. single identifiers, may be used as the type in declarations for variables, constants, and parameters. The language prohibits assignments and parameter passings in which the source and target do not have identical type names. Finally the operations defined for a type are those of its definition, i.e., for a matrix, the operations are those of array 1..n,1..n of real.

There is no way to avoid a defined type's inheriting the defining type's operations. Given the type equivalence criterion and the requirement that the type of actual parameters be equivalent to that of their formals, without these inherited operations, it would be impossible to define any operation on the new type [Ten 1978]. Thus, in

```
type table = array 1..100 of integer
type stack = record tp:  integer,
                    stk:  table
            endrecord;
type stuck = record tp:  integer,
                    stk:  table
            endrecord;
var a:  stack,
    u:  stuck
```

stack and stuck are different types, a and u cannot be assigned directly to each other as in

```
a:=u  or  u:=a
```

but it is permissible to do

```
a.tp: = a.tp+1;
u.tp: = u.tp+1
```

If not, it would be impossible to write the various operations needed, e.g., push, pop, top, is_empty, etc.

However, this very inheritance of the defining type's operations permits subverting the intent of the type equivalencing rule. To perform the effect of u:=a which itself is not allowed, one need do only

```
u.tp: = a.tp;
u.stk: = a.stk
```

These assignments are allowed because the components are of identical types. Thus, the attempt to get the protection of distinct types for non-encapsulated defined types does not work.

There are some languages with even more complicated compromises. For example, the Green Department of Defense language proposal [DOD 1978b] has one in which for only some of the type constructors, does each distinct use make a new type; for others, all uses with the same actual parameters make the same type. Such an approach suffers from the possibilities for subversion as well as the added complexity of nonuniformity.

The best approach for providing unencapsulated type definitions in a language permitting but not requiring encapsulation is the Euclid approach which may be characterized as an orthogonal combination of the Algol 68 approach and encapsulation. Simply stated:

1. All built-in types are distinct.
2. Each encapsulated (non-parameterized) type definition introduces a type distinct from all others.
3. For types introduced by use of constructors, e.g. array of, by use of type defining equations, e.g. type t1=t, or by a combination thereof, the equivalence is decided as in Algol 68. That is, two types are equivalent if and only if their definitions, understood recursively, are the same.

In other words, encapsulation must be used to obtain a distinct type with full protection against inappropriate operator application. Apart from encapsulation, type definition by equation (as one might expect from the use of the equal sign) produces totally equivalent types with no pretense of any protection. Thus, there is no surprise when protection is not obtained.

A Digression

A footnote in the introduction pointed out that the Pascal definition neglected to deal with the issue of type equivalence. At the same time, the folklore is replete with claims of language and definitional simplicity for Pascal (see for example [WSH 1977, GMK 1976]). Such claims for simplicity seem to miss the point. A language's simplicity has to be measured on the basis of its primitive features, its composition rules and its exceptions to the rules -- not necessarily on the readability of its definition. If that readability is obtained at the expense of completeness, complexity is in fact added to the language, as it is not certain what is the language. This leads to various dialects of the language being created, all under the same name -- clearly a complex situation.

Conclusion

We have seen that in the absence of complete encapsulation of data types, some form of structural equivalence of data types must be adopted to avoid type insecurities. It has been shown that the Euclid method of introducing optional encapsulation within a type system with structural equivalence is a viable way of achieving type security and the ability to introduce true abstract data types without excessive overhead.

References

[BEL 1977] Berry, D.M., Z. Erlich, C. Lucena, "Pointers and Data Abstractions in High-Level Languages - I: Language Proposals," Journal of Computer Languages, Vol. 2, pp. 135-148, 1977.

[DOD 1978a] Department of Defense, Red Programming Language Specification, February, 1978.

[DOD 1978b] Department of Defense, Green Programming Language Specification, February, 1978.

[GMK 1978] Goodenough, J.B., C.L. McGowan and J.R. Kelly, "Evaluation of Algol 68, Jovial J3B, Pascal, SIMULA 67, and TACPOL vs. TINMAN Requirements for a Common High Order Programming Language," Softech Inc., December, 1976.

[HoW 1973] Hoare, C.A.R., N. Wirth, "An Axiomatic Definition of the Programming Language Pascal," Acta Informatica, 2, 1973.

[JeW 1975] Jensen, K., N. Wirth, Pascal User Manual and Report, Springer Verlag, Berlin, 1975.

[Lam 1977] Lampson, B., et al., "Report on the Programming Language Euclid," SIGPLAN Notices, 12:2, 1977.

[LiZ 1974] Liskov, B., S. Zilles, "Programming with Abstract Data Types," SIGPLAN Symposium on Very High Level Languages, SIGPLAN Notices, 9:5, 1974.

[Pop 1977] Popek, G., et al., "Notes on the Design of Euclid," Proceedings of the ACM Conference on Language Design for Reliable Software, SIGPLAN Notices, Vol. 12, No. 3, 1977.

[Ten 1978] Tennent, R., "Another Look at Type Compatibility in Pascal," Software Practice and Experience, Vol. 8, pp. 429-437, 1978.

[vWi 1975] van Wijngaarden, et al., Editor, "Revised Report on the Algorithmic Language Algol 68," Acta Informatica, 5, 1975.

[WSH 1977] Welsh, J., J. Sneeringer, C.A.R. Hoare, "Ambiguities and Insecurities in Pascal," Software Practice and Experience, Vol. 7, pp. 685-696, 1977.