The Lattice of Data Types, or Much Ado about NIL

Andrew Black and Norman Hutchinson

Abstract

Some programming languages, notably object-oriented languages, structure their data types into a tree, with the child relationship indicating a more powerful type, i.e., one with more operations. This tree has been generalized, e.g., by Ingals and Borning [Borning and Ingalls 82], to a directed acyclic graph.

The present paper proposes a further requirement: that data types form a lattice under the same ordering. Bottom in the lattice is the root of the conventional inheritance tree, type Any with the minimal set of operations. Top is type None, possessing all possible operations. The unique element of this type is the familiar Nil, the entity which can be treated as belonging to any type, but which breaks as soon as one tries to operate on it.

1 Introduction

This paper is concerned with strongly typed programming languages and with the clear and uniform treatment of the indispensable constant Nil. It is motivated by our experiences with the Emerald programming language, but most of the ideas carry over in to a conventional strongly-typed language like Pascal or Algol-68.

Emerald is an object oriented language; its object structure is described in [Black et al. 86] and its type system in [Black et al. 87]. Each identifier in an Emerald program is associated at compile time with an abstract type, which is nothing more than a set of operations and their signatures. An operation signature gives the types of the arguments and results of the operation. The type of an identifier can be regarded as a statement of all the possible operations that can be performed on the objects that will be bound to it. For example, consider the type Directory defined as:

const *Directory* == **type** *T*
    Lookup[*K*: **String**] → [*E*: *Entry*]
    Add[*K*: **String**, *E*: *Entry*] → []
    Delete[*K*: **String**] → []
end *T*

This says that Directory is a type with three operations: Lookup, Add, and Delete. The signature of Lookup is "[*K*: **String**] → [*E*:*Entry*]", which means that it takes an argument of type String and returns a result of type Entry. Both the argument and result lists may be empty.

If a variable is declared as

var *d*: *Directory*

then it is legal to write invocations like *d.Lookup*[*"key"*] and *d.Delete*[*"bad"*] but not *d.New*. Type safety requires that when we execute the assignment

$d \leftarrow exp$

the object returned by exp does indeed support the operations Lookup, Add, and Delete, with the correct argument and result types. Moreover, we have chosen that in the normal case it is the responsibility of the compiler to ensure that this is true. It is perfectly acceptable for exp to return an object with additional operations, but those operations will never be invoked on d.

Slightly more formally, we say in Emerald that a object o can be bound to identifier i if the type of o conforms to the type of i. A type O conforms to type I, written O ⊳ I, iff:

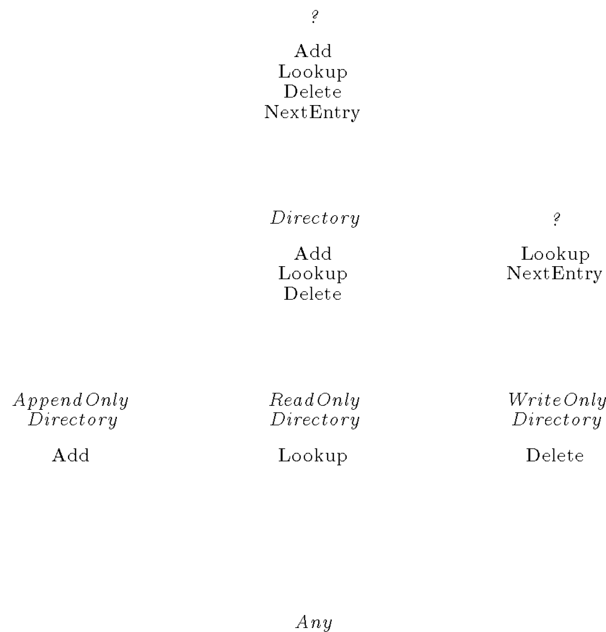
- All operations in I are also in O (with the same names)
- For each operation Ω_I and corresponding operation Ω_O
  - they have the same number of arguments
  - they have the same number of results
  - the type of each result of Ω_O conforms to the corresponding result of Ω_I
  - the type of each argument of Ω_I conforms to the corresponding argument of Ω_O.

Note that the results of the operations must conform in the same direction as the types to which they belong, but that the arguments must conform inversely. This reflects the opposing information flows of arguments and results.

In Emerald, the type Any has no operations. So, if a variable is declared as

var *a*: **Any**

any object at all can be assigned to a, since any object conforms to Any. The type ReadOnlyDirectory, with Lookup as its only operation, conforms to Any, and Directory conforms to ReadOnlyDirectory. In fact, ⊳ induces a partial order on types, as depicted in the following DAG:



Some types are incomparable: the type with Add as its only operation is incomparable to the type with Delete. Moreover, the type with *Add*[**String**] → [] is incomparable to the type with *Add*[**Integer**] → [**Integer**].

One problem with this scheme is how to type Nil, the name of the "undefined" object. We typically wish to use Nil in assignments

d := **Nil**

and in tests:

if $d \neq$ **Nil then** *d.Lookup*[*"key"*] **endif**

From what has been said so far, for Nil to be assignable to d, it must support all the directory operations. Similarly, for Nil to be assignable to

var i: **Integer**

it must support all of the integer operations. By extension, Nil must possess all of the operations of all possible types that might ever be constructed. This seems like a contradiction, because operationally we know that Nil does nothing. Indeed, it is a contradiction, because Nil would have to support add with an unbounded number of conflicting signatures.

The conventional solution to this dilemma, and the one that was adopted initially in the design of Emerald, is to make Nil a special case. Either Nil refers to a single special object that does not otherwise fit into the type system, or, as in Algol 68, there is a separate Nil for each reference type, and a syntactic mechanism for disambiguating the symbol Nil that denotes them all [vanWijngaarden et al. 76, Section 2.1, 3.2].

2 From partial order to lattice

The alternative solution presented in this paper is to accept the contradictions and to complete the design of the type system in such a way as they cause no damage. In the partial order we have so far described, it is the case that every pair of types T and U have a "meet" or greatest lower bound "T $\sqcap$ U" that contains just those operations that are common to T and U. If there are no operations in common the meet is Any, which is the bottom element in the partial order. If T and U have in common just operation α $[f] \rightarrow [g]$ then the type containing just α with that signature is their meet. However, if they both support operations α but with different signatures α $[f_T] \rightarrow [g_T]$ and α $[f_U] \rightarrow [g_U]$, then T meet U is the type containing α $[f_T \sqcup f_U] \rightarrow [g_T \sqcap g_U]$ if $f_T \sqcup f_U$ (the join of f_T and f_U) is defined, and Any otherwise. This definition generalizes in the obvious way to types with more than one operation.

The conventional mathematical solution to this inelegant caveat "if the join exists" is to declare that all joins do exist, i.e., to embed the partial order in a complete lattice. We do not loose generality by this assumption, since such a lattice always exists [Stoy 77, pp. 88-91, 414]. What is a little surprising is that (at least some of the) the extra elements we introduce are semantically useful. In particular, the top element of this lattice of types provides a type for Nil.

We define T $\sqcap$ U as the largest type such that T $\circ\!>$ T $\sqcap$ U and U $\circ\!>$ T $\sqcap$ U, and T $\sqcup$ U as the smallest type such that T $\sqcup$ U $\circ\!>$ U and T $\sqcup$ U $\circ\!>$ T. Just as we denote bottom ($\perp$), the minimal type s.t. for all types T, T $\circ\!>$ $\perp$ by Any, we denote top ($\top$), the maximal type s.t. for all types T, $\top$ $\circ\!>$ T by None. Nil is simply an object of type None; it supports all possible operations, with all possible signatures, including the contradictory ones. In other words, Nil supports add with 1,3, and 17 arguments of all possible types. It supports an operation miracle s.t. wp [miracle] R = true and an operation *Halt*[*Turing-machine, Tape*] $\rightarrow$ [*Boolean*]. Of course, we are speaking only of type checking, which we view as syntactic. If an attempt is made to invoke any of these operations, the implementation breaks, which is exactly the characteristic one expects of Nil.

3 A lattice of types or a lattice of signatures

The preceding discussion has demonstrated one benefit of extending the partial order of types to a complete lattice — the usefulness of the top element, None. We are left with the problem of defining the join (or least upper bound) operation on types. In the absence of overloading (which would require the modification of our notion of type to allow a type to include multiple operations with the same name, and which we have systematically ignored in this discussion), we have two obvious choices.

Since our motivation for forming a lattice from the partial order of types was this lattice's top element ($\top$ or None), we could define the join of two types with conflicting operation signatures to be $\top$ in every case. That is, if types T and U are defined by:

type T
  $A[\mathit{Integer}] \rightarrow [\mathit{Integer}]$
  $B[\textbf{Integer}] \rightarrow [\textbf{Real}]$
end T

type U
  $A[\mathit{Integer}] \rightarrow [\mathit{Integer}]$
  $B[\textbf{Integer}, \textbf{Real}] \rightarrow []$
end U

Then we define T $\sqcup$ U as $\top$ because the signatures of B_T and B_U contradict. In defining $\sqcup$ in this way we make minimal changes to our original partial order, adding only the single element None in order to complete the lattice.

Our second alternative is to allow multiple contradictory types instead of collapsing all of them to the single element $\top$. Before getting into the details, let us first consider an example. Given types T and U above, we see that their B operations conflice, but their A operations are compatible, in fact identical. It may be useful to define their join ($\sqcup$) as a type with an operation A from Integers to Integers, and an operation B with a contradictory signature, as follows:

type $TjoinU$
  $A[\mathit{Integer}] \rightarrow [\mathit{Integer}]$
  $B\ \top$
end $TjoinU$

This definition of join retains what information it can from the two types being joined, while adequately reflecting those operations on which the conflict.

To formally define the join to achieve this effect, we define a lattice of operation signatures, including both a top and bottom element. The bottom element of this lattice was previously useful in specifying that the operation is non existent in a type [Black et al. 87]; the top element is now useful as the specification of a contradictory operation. We can now define the join operation between operation signatures, o and p.

- If either of o or p is $\bot$, then the join is the other.
- If either is $\top$ then the join is $\top$.
- If o and p have differing number of arguments or results then the join is $\top$.
- If o and p have the same numbers of arguments and results, the the join is the signature with the pairwise $\sqcap$ ($\sqcup$) of the corresponding argument (result) types.

4 Emerald

The Emerald type system was designed as a partial order, and later extended to a lattice by the introduction of the single top element None. All contradictory types therefore collapse into this single element.

5 Comparison with Other Work

The idea of expressing the relationship between data types as a lattice has a long history. However, the present work is (to our knowledge) unique in that it deals with types as sets of operations rather than as sets of values, as advocated for example by Donahue and Demers [Donahue and Demers 85].

Cousot and Cousot [Cousot and Cousot 77] address the problem of positioning Nil in a lattice of reference types. However, the focus of their paper is to eliminate the run-time check that is otherwise necessary before dereferencing a pointer. They treat each conventional reference type as the top of a lattice of types containing the singleton type of Nil and the type of non-Nil references as the only proper elements. They thus require that there be a separate Nil pointer for each reference type. Their ordering

relation is inclusion of values: the values of the complete conventional reference type is the union of Nil and the non-Nil reference values.

In the same year, Shamir and Wadge [Shamir and Wedge 77] presented a type system in which all types and all values are part of the same lattice. Their ordering relation is the conventional approximation ordering on values, and inclusion on types; the assertions x $\sqsubseteq$ y, x is of type y, and any object of type x is of type y are all equivalent. So, for example, all the integer values and truth values are incomparable, but 2 and 4 are both $\sqsubseteq$ eveninteger, which is $\sqsubseteq$ integer which is $\sqsubseteq$ real. As in the Emerald type system, the question "what is the type of 5" is meaningless: any particular element of their lattice has a chain of types that describe it to a lesser or greater degree of accuracy. In Emerald, this chain is finite, because all obejcts have a finite set of operations. The l.u.b. of this chain contains all of the operations; we call it the "best fitting abstract type". In Shamir and Wadge's system, since a type is a set of values, each value belongs to an infinite set of intuitive types. 5 is of type integer, 5 is of type real, 5 is of type prime, $\perp$ is of 5 and 5 is of type U (the Universal type) are all true assertions.

The treatment of functions in Shamir and Wadge's system is somewhat similar to Emerald. In particular, they recognise the essential antimonotonicity of argument types. real $\sqsupseteq$ integer, since real contains all the integer values; integer $\rightarrow$ real (the type of all functions from integers to reals) $\sqsupseteq$ integer $\rightarrow$ integer, which in turn $\sqsupseteq$ real $\rightarrow$ integer.

A recent paper by Cardelli, Donahue and Nelson [Cardelli et al. 87] describes a type system with the most similarity to that advocated here. Modula 3 aims to clean up the ragged edges of Modula 2's type system where it deals with subranges and opaque types, and also introduces a simple set of object types with inheritance. A notion of subtyping is introduced that orders types by containment, although this is interpreted in such a way that it also provides for a weak conformity of object types.

In Modula 3, a type T is a subtype of U, written T <: U, if every value of type T is also a value of type U. This is motivated by the example [0 .. 9] <: Integer. If t is of type T, and u is of type U, then the assignment u := t is safe, and t := u requires (in general) a run-time check. (Modula 3 makes this check implicit.) The notion of containment of values is extended so that

```
TYPE IC = REF RECORD i: INTEGER; c: CHAR END;
TYPE I = REF RECORD i: INTEGER END;
```

are ordered by IC <: I. This is motivated by the fact that an IC value is the address of an integer field i, followed by a char field c, whereas an I value is the address of an integer field i. Every value of type IC is therefore also a value of type I, from which the stated ordering follows by definition.

In Emerald, records are considered to be a shorthand for objects with the appropriate Set and Get operations to access the fields. So type I has SetI and GetI operations that accept and return an integer, and type IC has these operations as well as additional operations to access c. In our ordering, I $<\!\circ$ IC (read I has fewer operations that IC, or IC conforms to I). Both systems allow an IC value to be assigned to an I variable, but the inverse assignment requires a run-time check (which is implicit in Modula 3, and explicit in Emerald — a view expression.)

Beneath this superficial simlarity, the two type systems are very diferent. Basically, Emerald's concern with operations rather than values yields a type system that is far more abstract. Type conformity is not concerned with the order of the fields in a record, or indeed whether they exist at all. Two quite different representations can both conform to the same type, provided that they possess the same operations. The arguments and results of the operations need not be identical, provided that they conform in the appropriate way.

In contrast, Modula 3's preoccupation with the concrete details of an object's representation permits a simple form of inheritance based on adding fields and operations to an existing type. In Emerald's more abstract setting, it is not even clear how to give a meaning to the question "is every value of type T also a value of type U?" Comparing the bits that make up a T and seeing if they can be interpreted as a U is inadequate, and not only because both T and U can have multiple, quite different, represenations. By 'is every T also a V', the Modula 3 type system really means to ask 'when a T value is interpreted as a U value, is it *the same* value? Emerald views sameness a *semantic* issue, and as such quite outside the jurisdiction of the type system.

In fact, the only Emerald types for which the "value containment" definition of typing would make sense are those built-in types which may not be reimplemented, whose representaion is known to the

compiler, and whose semantics are part of the language definition. Chief among these are the integers. Emerald does not provide a build in constructor for integer subranges, but if it did, in order to make [0 .. 9] $\circ\!\!>$ Integer, but not Integer $\circ\!\!>$ [0 .. 9], the subrange would have to have more operations than Integer. To make this possible in general, it would be necessary to invent maxinteger^2 artificial operations, solely in order to provide the proper inclusion on integer subtypes. An explicit create operator would be needed to achieve the conversion from integer to each subrange.

In Modula 3, Null (the type containing the single value Nil) is treated as <: all reference, pointer and object types. This implies that (the reresentation of) Nil is also a (representation of the same) value in all of those types, which is indeed the conventional interpretion of Nil in a language like Pascal. However, Modula 3, along with Emerald, achieves this in a unified and consistent setting.

6 Summary

Nil has clean semantics, but the irony of the situation is that no one cares!

References

[Black et al. 86] Andrew Black, Norman Hutchinson, Eric Jul, and Henry Levy. Object structure in the Emerald system. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 78–86, ACM, October 1986. Published in SIGPLAN Notices, vol. 21, no. 11, November 1986.

[Black et al. 87] Andrew Black, Norman Hutchinson, Eric Jul, Henry Levy, and Larry Carter. Distribution and abstract types in Emerald. *IEEE Transactions on Software Engineering*, 13(1), January 1987.

[Borning and Ingalls 82] Alan H. Borning and Daniel H. H. Ingalls. *Multiple Inheritance in Smalltalk80.* Technical Report TR 82-06-02, UWCS, June 1982.

[Cardelli et al. 87] L. Cardelli, J. Donahue, and G. Nelson. *The Module 3 Type System.* Draft Technical Report, DEC SRC, October 1987. Appendix to Draft Modula 3 Language Specification.

[Cousot and Cousot 77] Patrick Cousot and Radhia Cousot. Static determination of dynamic properties of generalized type unions. *Proceedings of ACM Conference on Language Design for Reliable Software*, 77–94, March 1977. SIGPLAN Vol 12, Nr 3.

[Donahue and Demers 85] James Donahue and Alan Demers. Data types are values. *ACM Transactions on Programming Languages and Systems*, 7(3):426–445, July 1985.

[Shamir and Wedge 77] Adi Shamir and William Wedge. Data types as objects. In *Lecture Notes in Computer Science, Volume 52 - Automata, Languages and Programming*, Springer-Verlag, 1977. pp. 465-479.

[Stoy 77] Joseph E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* MIT Press, 1977.

[vanWijngaarden et al. 76] Adriaan van Wijngaarden, B.J. Mailloux, J.E.L. Peck, C.H.A. Koster, M. Sintoff, C.H. Linsey, L.G.L.T. Meertens, and R.G. Fisker. *Revised Report on the Algorithmic Language Algol68.* Springer Verlag, 1976.