TECHNICAL CORRESPONDENCE

Type-Extension Type Tests Can Be Performed In Constant Time

NORMAN H. COHEN
IBM Thomas J. Watson Research Center

Wirth's proposal for type extensions includes an algorithm for determining whether a given value belongs to an extension of a given type. In the worst case, this algorithm takes time proportional to the depth of the type-extension hierarchy. Wirth describes the loop in this algorithm as "unavoidable," but in fact, the test can be performed in constant time by associating a "display" of base types with each type descriptor.

Categories and Subject Descriptors: D.3.3 [**Programming Languages**] Language Constructs and Features—*datatypes and structures, procedures, functions, and subroutines*; D.3.4 [**Programming Languages**] Processors—*code generation, compilers, run-time environments*; E.1 [**Data**] Data Structures—*lists*; E 2 [**Data**] Data Storage Representations

General Terms: Algorithms, Languages, Performance

Additional Key Words and Phrases: Class, descriptor, display, extensible data type, inheritance, membership test, object-oriented programming, type extension, type test

Wirth [2] proposes a programming-language feature called *type extension*. A declaration of the form

```
LT' = RECORD(T) . . . END
```

extends the previously declared type T to form a new type T′. If, for example, T is a record type, then T′ is a record type having all the fields of T plus the fields declared at the point of the ellipsis. Type T′ is called a *direct extension* of type T, and type T is called the *direct base type* of type T′. A type t_1 is an *extension* of type t_2 if $t_1 = t_2$ or t_1 is a direct extension of an extension of t_2; a type t_2 is a *base type* of a type t_1 if $t_2 = t_1$ or t_2 is a direct base type of a base type of t_1. Furthermore, a pointer type p_1 is an extension of a pointer type p_2 if the type pointed to by p_1 is an extension of the type pointed to by p_2.

Author's address: IBM Research, Thomas J. Watson Research Center, P.O Box 704, Yorktown Heights, NY 10598.

A value may be assigned to a variable if and only if the type of the value is an extension of the type of the variable. The assignment discards any fields of the value that are not fields of the variable. When the value or variable in an assignment is referenced by a pointer, a run-time check may be necessary to determine type compatibility. A VAR formal parameter is considered to be an implicit reference by pointer to the actual parameter, so a run-time type-compatibility check may also be required for uses of VAR formal parameters. Wirth proposes that each variable referenced by a pointer (including actual VAR parameters) be accompanied by a hidden pointer to a *type descriptor* for the type of the variable. The hidden pointer to the type descriptor is called a *type tag*. The descriptor for an extended type includes a "basetag" field pointing to the descriptor for the direct base type. The following loop determines whether a value v may be assigned to a variable of type TO:

```
t := v.tag;
LOOP
  IF t = TO THEN EXIT (TRUE) END;
  t := t↑.basetag;
  IF t = NIL THEN EXIT (FALSE) END
END
```

Wirth writes, "Unfortunately, the provision of a loop construct for descending the linked list of tags is unavoidable since in the context of the test no information is available about the number of extensions of the tested type that may have been defined in other modules." In fact, sufficient information is available in the context of the test to make the loop unnecessary. This information comprises the set of base types of the value whose type is being tested, the depth of each of these base types in the type-extension hierarchy, and the depth of the tested type in the type-extension hierarchy.

The loop can be eliminated by an adaptation of the "display" originally proposed by Dijkstra [1] to address nonlocal variables in a block-structured language. The display can be thought of as a linear-list representation of a linked list whose length is known. In Dijkstra's scheme, directly indexing an element of this linear list obviates the need to traverse a linked list of static environment pointers.

In our adaptation, the descriptor for a given type includes a display containing the type tags of each of its base types. Directly indexing an element of this display obviates the need to traverse a linked list of type descriptors. Unlike the displays used to address global variables, the displays used for type tests can be constructed entirely at compile time.

Define the *depth* of a type in the type-extension hierarchy as follows: A type that is neither an extended type nor a pointer type has depth zero; a direct extension of a type with depth n has depth $n + 1$; and a pointer type has the same depth as the type to which it points. Then a type at depth n in the type-extension hierarchy has a unique base type at each depth d, $0 \leq d \leq n$. (For a pointer type pointing to some type t, the base type at a given depth will be a type pointing to some base type of t, even though such a type may not have been explicitly declared.) The descriptor for a type at

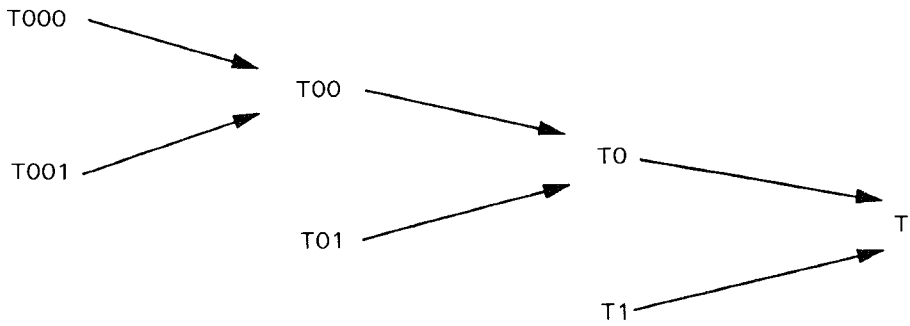

Fig. 1. Family of types related by type extension An arrow from type a to type b means that type a is a direct extension of type b. The base types of a given type are found along the path from that type to the root of the tree.

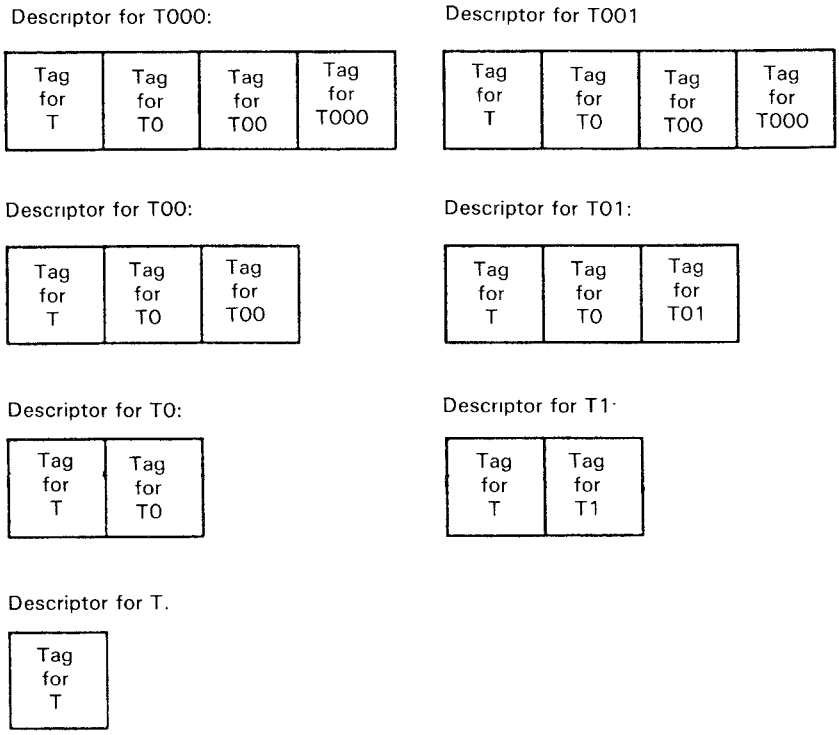

Fig. 2. Displays contained in the descriptors for each of the types shown in Figure 1. The display for a given type lists all of the type's base types, indexed by distance from the root of the tree in Figure 1 A "tag" for a given type is a pointer to the descriptor for that type

depth n includes a display with elements indexed from 0 to n. The element of the display indexed by d is a type tag pointing to the descriptor for the type's base type of depth d, $0 \leq d \leq n$.

The compiler constructs a descriptor for each type as it encounters the declaration of that type. For a type that is not declared as a direct extension

of some other type, the descriptor contains a display with one element, pointing to the descriptor itself. A type may be declared as a direct extension of some other type only after that other type has been declared (perhaps in some module compiled earlier). Thus, the compiler has access to the descriptor of the direct base type. The descriptor of the type extension is constructed with a display consisting of the display for the direct base type followed by a pointer to the new descriptor itself.

Figure 1 depicts the type-extension hierarchy used in the examples of [2]. Figure 2 illustrates the displays of each of the seven types illustrated in this hierarchy.

Assume that the display included in a type descriptor *td* is implemented by two record components, *td*.depth and *td*.base_types. Component *td*.depth gives the depth of the described type in the type-extension hierarchy. Component *td*.base_types is an array of type tags indexed from 0 to *td*.depth, with *td*.base_types[*i*] pointing to the descriptor for the base type at depth *i* of the type described by *td*, $0 \leq i \leq td.\text{depth}$. Then the test to determine whether some dynamically allocated variable or VAR parameter *v* belongs to an extension of type TO is as follows:

```
  t := v.tag
IF t↑.depth < TO↑.depth THEN
  RETURN FALSE
ELSE
  RETURN t↑.base_types[TO↑.depth] = TO
END
```

The first arm of the IF statement handles the case in which type TO has a greater depth than type *t*, and so cannot possibly be a base type of type *t*. In the second arm of the IF statement, *t*↑.base_types is a list of the base types of type *t*. If the type tag for type TO is on this list, it must be at position TO↑.depth.

REFERENCES

1. Dijkstra, E. W. Recursive programming. *Numer. Math. 2* (1960), 312–318.
2. Wirth, N. Type extensions. *ACM Trans. Program. Lang. Syst. 10*, 2 (Apr. 1988), 204–214.

Received December 1988; accepted January 1990