Towards An Agreeable Model of Type Inheritance

Hugh Darwen
IBM United Kingdom Limited
Hugh_Darwen@uk.ibm.com

November 1999

References

- Hugh Darwen: "Towards An Agreeable Model of Type Inheritance", submitted to *Journal of Computer Science and Information Management*, September 1999 issue.

- C.J. Date and Hugh Darwen: "Foundation for Future Database Systems: *The Third Manifesto*", Addison-Wesley, 2000 (to appear, as 2nd edition of "Foundation for Object/Relational Databases: *The Third Manifesto*", Addison-Wesley, 1998).

Why Seek a Model of Type Inheritance?

- Because there doesn't seem to be (a commonly agreed) one.
- Because what purports to be type inheritance in (e.g.) Smalltalk, C++, Java, SQL:1999 seems to be *ad hoc* (and, as type inheritance, not very agreeable).
- For relational databases, we wanted a rigorous definition, in keeping with relational theory.

Assumptions

The language ("D") embraces these concepts:

- **Value**
  - 3
  - 'All logical differences are big differences'
  - POINT (3 , 4)
  - {1.5, -3.7, 14.0}
- **Variable**
  - to which exactly one value is assigned
  - which might vary from time to time
- **Operator**
  - *read-only* operator, on values, yielding a value when invoked
  - *update* operator, on values and at least one variable (e.g., assignment), yielding no value
- **Type**
  - *declared* type, of a variable, parameter, operator invocation
  - *most specific* type, of a value

and conforms to *The Third Manifesto!*

Conformance to *The Third Manifesto*

- D supports type generators for tuple types and relation types, along with prescribed operators on tuples and relations.
- D (unlike SQL) supports a truth-valued type consisting of the values *true* and *false*, and is firmly based in two-valued logic.
- D (unlike SQL) exhibits *conceptual integrity* (slavish adherence to stated concepts).
- D (unlike SQL) adheres to generally agreed language design principles.
- SQL's mistakes are not made in D (so, e.g., no nulls, no duplicate rows, no anonymous columns, column names unique within table, "=" means equals and tables with no columns are recognised)

Terminology

- Subtype, supertype
  If T2 is a subtype of T1, then every value in T2 is a value in T1.

- Proper subtype, supertype
  T is a subtype (supertype) of T, but is not a proper subtype (supertype) of T.

- Immediate subtype, supertype
  T2 is an immediate subtype of T1 *iff* T2 is a subtype of T1 and no type is both a proper subtype of T1 and a proper supertype of T2

- Root type
  has no proper supertypes

- Leaf type
  has no proper subtypes

Restrictions and Non-restrictions

- A value has one most specific type, which is not necessarily a leaf type.
- If every value in T is of some proper subtype of T, then T has at least two immediate subtypes and is called a *union type*.
- A type can have more than one immediate supertype (multiple inheritance).
- Multiple immediate supertypes of T must have a common supertype.
- The values constituting type T (including its subtypes!) are specified by a *possible representation* and a *constraint*. (Not necessary if T is a union type. A union type with no "possrep" is a *dummy type*.)

Running Examples

```
TYPE ELLIPSE
  IS PLANE_FIGURE
    POSSREP { A LENGTH, B LENGTH, CTR POINT
                CONSTRAINT A >= B AND B > LENGTH ( 0.0 ) } ;

TYPE CIRCLE
  IS ELLIPSE
    CONSTRAINT THE_A ( ELLIPSE ) = THE_B ( ELLIPSE )
    POSSREP { R = THE_A ( ELLIPSE ) ,
                  CTR = THE_CTR ( ELLIPSE ) } ;


TYPE SQUARE
  IS { RECTANGLE, RHOMBUS }
    POSSREP ... ;  /* no extra constraint needed! */
```

POSSREP declaration implies certain operator definitions. "Selectors" are akin to OO constructors. Counterparts of observers and mutators too.

Two More Assumptions

(and they are important ones!)

1. Declared type constraints are consistent with our model.
   e.g., if RECTANGLE and RHOMBUS are leaf types, then there is no parallelogram that is both a rectangle and a rhombus! To reflect reality, type SQUARE *must* therefore be defined.

2. If different versions exist of the same operator, they have identical effects when given identical operands.
   e.g., if C is a circle, AREA(C) can be evaluated by executing either the general ellipse version of AREA or the special version for circles. Performance might vary, though!

We don't legislate for these. We merely say that consequences of violation are unclear.

Salient Features of Our Model

- Value substitutability
  Wherever a value of type ELLIPSE is expected, a value of type CIRCLE can be given. *Note:* there is no principle of *variable* substitutability!

- Static type checking
  Type mismatches can be caught by inspection of source code. (One prescribed operator, "TREAT DOWN", can give run-time type checks, as in SQL:1999).

- "Mutability"
  i.e., assignment (and possibly other update operators)

- Specialisation by constraint
  e.g, an ellipse with A=B *is* a circle of radius R (=A=B), even if it results from invocation of the ELLIPSE selector as opposed to CIRCLE selector.

The above in spite of Maier and Zdonik's "3 out of 4" conjecture, that a system can support any 3 of these but not all 4!

The Three Sections of Our Definition

1. Single inheritance with scalar types

2. Multiple inheritance

3. Tuple and relation type inheritance
   (requires multiple inheritance, because, e.g.,
   a tuple of type
   TUPLE { E1 CIRCLE, E2 CIRCLE }
   is of type
   TUPLE { E1 ELLIPSE, E2 CIRCLE }
   and also of type
   TUPLE { E1 CIRCLE, E2 ELLIPSE },
   neither of which is a supertype of the other!)

Two Very Special Scalar Types

alpha is the conceptual dummy type that is a proper supertype of every scalar type except itself and a proper subtype of none.

Every value is a value of type *alpha*.

Equals comparison of values is defined for *alpha* and therefore for every type. Assignment is also defined for *alpha*.

omega is the conceptual dummy type that is a proper subtype of every scalar type except itself and a proper supertype of none.

Every scalar read-only operator is therefore defined for values of type *omega*.

No value is a value of type *omega!*

An Interesting Question

Let SE be an expression of declared type SET (ELLIPSE) and let SC be an expression of declared type SET (CIRCLE).

What is the declared type of

SE INTERSECT SC ??

Should it be the same as the declared type of

SE UNION SC
MINUS
((SE MINUS SC) UNION (SC MINUS SE)) ??

If so, the answer must be SET (ELLIPSE)!

Computability of Most Specific Type

O-O pundits tend to reject specialisation by constraint because of the claimed unbearable overhead of computing the MST "every time an object is touched".

We dispute this claim.

It is *never* necessary to compute the MST.

It is only necessary (sometimes) to test at run time if a given value has a some type ST that is more specific than the declared type DT of the expression denoting it, in the case where specialised versions of a required operator exist.

And even then it is not necessary to visit every type between DT and ST.

A Nasty Consequence of Oids

Let E be a Java variable of type ELLIPSE.
Let C be a Java variable of type CIRCLE.

Consider these assignments:

1. C := new CIRCLE (5, POINT (0, 0));
2. E := C ;

3. E.A := 6;

Because new returns an oid, E and C are both assigned the same pointer, not the same ellipse.

Assignment 3 assigns to the A component of the ELLIPSE object pointed to by E.

But that is also the object C points to, no longer a circle!

A New "4 out of 5" Conjecture

Perhaps Maier and Zdonik assumed the existence of objects (with oids).

In that case, we suggest that a type system can embrace any *four* of:

1. Value substitutability.
2. Static type checking.
3. "Mutability".
4. Specialisation by constraint.
5. Objects (with oids).

but not all five.

Happily, our choice was made for us in 1969, by E.F. Codd!

The End