The Inheritance Model

(version dated September 26th, 2016, superseding all previous versions)

O England! model to thy inward greatness,
Like little body with a mighty heart,
What mightst thou do, that honour would thee do,
Were all thy children kind and natural!

—William Shakespeare:
King Henry the Fifth (1598-1599)

What follows is Chapter 4 of *Type Inheritance and Relational Theory* by C. J. Date, O'Reilly, 2016 (hence the references in the introductory text to "this book", "this chapter", and suchlike).

The introductory text uses the first person singular to refer to the book's sole author, Chris Date, but the section THE IM PRESCRIPTIONS—i.e., the rest of the Chapter—is a joint work by him and Hugh Darwen.

Any further revisions to the Inheritance Model will appear in this document, which should then be taken as definitive, superseding Chapter 4 of *Type Inheritance and Relational Theory*.

This chapter provides, for purposes of subsequent reference, a precise statement of the 28 IM prescriptions that make up our inheritance model. It's based on Chapter 19 of the book Database Explorations: Essays on The Third Manifesto and Related Topics, *by Hugh Darwen and myself (available free online at the website www.thethirdmanifesto.com). However, I've found it necessary, or at least convenient, to perform a certain amount of revision on some of the prescriptions, as will be made clear in subsequent chapters. I've also added two new ones (numbers 23 and 26, according to the numbering below). Note: Whenever there's a technical discrepancy between the present chapter—or anything else in this book, come to that—and previous publications by Darwen and myself on this topic, the present text should be taken as superseding. At the same time, please note that it's my intention that any such discrepancies be called out explicitly and justified.*

Throughout this chapter, as well as elsewhere in this book, I use the symbols *T* and *T′* as generic names for a pair of types such that *T′* is a subtype of *T* (equivalently, such that *T* is a supertype of *T′*). You might find it helpful to think of *T* and *T′* as ELLIPSE and CIRCLE, respectively; however, keep in mind that they're not limited to being scalar types specifically, barring explicit statements to the contrary (moreover, the various prescriptions are all worded in such a way as not to be limited to single inheritance only, either). Note too that distinct types have distinct names; in particular, if *T′* is a proper subtype of *T*, then their names will be distinct, even if the set of values constituting *T′* isn't a proper subset of the set of values constituting *T*. (Conversely, if their names aren't distinct, then *T′* and *T* are the very same type and the corresponding sets of values will be identical.) Also, I assume that all of the types under discussion, including the maximal and minimal types discussed in IM Prescriptions 20 and 25, are members of some given set of available types *GSAT* (though the only explicit mention of that set is in IM Prescription 20, q.v.); in particular, the definitions of the terms *root type* and *leaf type* in IM Prescription 6 are to be understood in the context of that set. For example, given the type hierarchy of Fig. 3.1 in Chapter 3, the set of available types consists of:

a. PLANE_FIGURE, ELLIPSE, CIRCLE, POLYGON, RECTANGLE, and SQUARE

b. The types in terms of which the possreps for the types listed under point a. are defined

c. The types in terms of which the possreps for the types included under point b., such as LENGTH and POINT, are defined (and so on, recursively, all the way down to and including the pertinent primitive types—see below)

d. The maximal scalar type *alpha* and the minimal scalar type *omega* (see IM Prescription 20)

e. Tuple and relation types that can be generated using any of the types mentioned in any of these five points a.-e.

Note: The term *primitive type*, mentioned under point c. above, refers to a system defined type (scalar by definition) with no declared possrep. The qualifier *primitive* derives from the fact that all of the types available in any given context are ultimately defined in terms of such types. Typical examples of such primitive types include the types INTEGER, RATIONAL, CHAR, and BOOLEAN.

By the way, it's worth stating explicitly that type PLANE_FIGURE is *not* the only root type with respect to the foregoing set of types. It's not even the only scalar root type. By way of example, consider type POINT. Since it's the type of (among other things) a possrep component for type CIRCLE, type POINT is certainly a member of the given set of types; however, it's not a subtype of PLANE_FIGURE, and so it must be part of some distinct type hierarchy—possibly one consisting of type POINT only—and, by definition, that distinct type hierarchy has a distinct root type of its own.

THE IM PRESCRIPTIONS

1. *T* and *T′* shall each be types; i.e., each shall be a named set of values.

2. Every value in *T′* shall be a value in *T*; i.e., the set of values constituting *T′* shall be a subset of the set of values constituting *T* (in other words, if a value is of type *T′*, it shall also be of type *T*).

3. *T* and *T′* shall not necessarily be distinct; i.e., every type shall be both a subtype and a supertype of itself.

4. Every subtype of *T′* shall be a subtype of *T*. Every supertype of *T* shall be a supertype of *T′*.

5. Let *T* and *T′* be scalar types. Then:

   a. If and only if *T* and *T′* are distinct, then *T* shall be a **proper** supertype of *T′* and *T′* shall be a **proper** subtype of *T*.

   b. Let *T* be a proper supertype of *T′*, and let *S* be a sequence of types *T1*, *T2*, ..., *Tm* such that *T* is a proper supertype of *T1*, *T1* is a proper supertype of *T2*, ..., and *Tm* is a proper supertype of *T′* ($m \geq 0$). Then either (a) no such sequence *S* shall exist (i.e., every such sequence shall be such that $m = 0$), in which case (and in which case only) *T* shall be an **immediate** supertype of *T′*, or (b) every such sequence *S* shall be such that $m > 0$, in which case (and in which case only) *T* shall be a **nonimmediate** supertype of *T′*. Also, *T′* shall be an **immediate** subtype of *T* if and only if *T* is an immediate supertype of *T′*, and *T′* shall be a **nonimmediate** subtype of *T* if and only if *T* is a nonimmediate supertype of *T*.

   c. If and only if *T* is an immediate supertype of *T′* and *T′* is neither a root type nor type *omega*—see IM Prescription 20—then the definition of *T′* shall be accompanied by a specification of an **example value** that is of type *T* and not of type *T′*.

6. A scalar type that has type *alpha*—see IM Prescription 20—as its sole immediate supertype shall be a (scalar) **root** type. A scalar type that has type *omega*—again, see IM Prescription 20—as its sole immediate subtype shall be a (scalar) **leaf** type.

7. Types *T1* and *T2* shall be **disjoint** if and only if no value is of both type *T1* and type *T2*. Types *T1* and *T2* shall **overlap** if and only if there exists at least one value that is common to both. Distinct root types shall be disjoint. If types *T1* and *T2* are distinct immediate subtypes of the same scalar type *T*, there shall exist at least one value that is of type *T1* and not of type *T2*.

8. Let *T1*, *T2*, ..., *Tm* ($m \geq 0$), *T*, and *T′* be scalar types. Then:

   a. Type *T* shall be a **common supertype** for, or of, types *T1*, *T2*, ..., *Tm* if and only if, whenever a given value is of at least one of types *T1*, *T2*, ..., *Tm*, it is also of type *T*.

Further, that type *T* shall be the **most specific** common supertype for *T1*, *T2*, ..., *Tm* if and only if no proper subtype of *T* is also a common supertype for those types.

b. Type *T′* shall be a **common subtype** for, or of, types *T1*, *T2*, ..., *Tm* if and only if, whenever a given value is of type *T′*, it is also of each of types *T1*, *T2*, ..., *Tm*. Further, that type *T′* shall be the **least specific** common subtype—also known as the **intersection type** or **intersection subtype**—for *T1*, *T2*, ..., *Tm* if and only if no proper supertype of *T′* is also a common subtype for those types.

Given types *T1*, *T2*, ..., *Tm* as defined above, it can be shown (thanks in particular to IM Prescription 20) that a unique most specific common supertype *T* and a unique least specific common subtype *T′* always exist. In the case of that particular common subtype *T′*, moreover, it is required that whenever a given value is of each of types *T1*, *T2*, ..., *Tm*, it is also of type *T′* (hence the alternative term *intersection type*). And it can be shown as a consequence that every scalar value *v* has both a unique least specific type and a unique most specific type (regarding this latter—which elsewhere in these prescriptions is denoted *MST*(*v*)—see also IM Prescription 9).

9. Let scalar variable *V* be of declared type *T*. Because of value substitutability (see IM Prescription 16), the value *v* assigned to *V* at any given time can have any nonempty subtype *T′* of type *T* as its most specific type. We can therefore model *V* as a named ordered triple of the form <*DT*,*MST*,*v*>, where:

a. The name of the triple is the name of the variable, *V*.

b. *DT* is the name of the declared type for variable *V*.

c. *MST* is the name of the **most specific type**—also known as the **current** most specific type—for, or of, variable *V*.

d. *v* is a value of most specific type *MST*—the **current value** for, or of, variable *V*.

We use the notation *DT*(*V*), *MST*(*V*), *v*(*V*) to refer to the *DT*, *MST*, *v* components, respectively, of this model of scalar variable *V*. *Note:* Since *v*(*V*) uniquely determines *MST*(*V*)—see IM Prescription 8—the *MST* component of *V* is strictly redundant. We include it for convenience.

Now let *X* be a scalar expression. By definition, *X* represents an invocation of some scalar operator *Op*. Thus, the notation *DT*(*V*), *MST*(*V*), *v*(*V*) just introduced can be extended in an obvious way to refer to the declared type *DT*(*X*), the current most specific type *MST*(*X*), and the current value *v*(*X*), respectively, of *X*—where *DT*(*X*) is the declared type of the invocation of *Op* in question (see IM Prescription 17) and is known at compile time, and *MST*(*X*) and *v*(*X*) refer to the result of evaluating *X* and are therefore not known until run time (in general).

10. Let *T* be a regular type (see IM Prescription 20) and hence, necessarily, a scalar type, and let *T′* be a nonempty immediate subtype of *T*. For each such immediate supertype *T* of *T′*, the definition of *T′* shall specify a **specialization constraint** *SC*, formulated in terms of *T*, such that a value shall be of type *T′* if and only if it satisfies all such constraints *SC*.

11. Consider the assignment

    ```
    V := X
    ```

    (where *V* is a variable reference and *X* is an expression). *DT*(*X*) shall be a subtype of *DT*(*V*). The assignment shall set *v*(*V*) equal to *v*(*X*), and hence *MST*(*V*) equal to *MST*(*X*) also.

12. Consider the equality comparison

    ```
    Y = X
    ```

    (where *Y* and *X* are expressions). *DT*(*Y*) and *DT*(*X*) shall overlap. The comparison shall return TRUE if *v*(*Y*) is equal to *v*(*X*) (and hence if *MST*(*Y*) is equal to *MST*(*X*) also), and FALSE otherwise.

13. Let *RX* and *RY* be relational expressions. In accordance with IM Prescription 28, each of *RX* and *RY* has a declared type. Let those declared types have headings

    ```
    { <A1,TX1> , <A2,TX2> , ... , <An,TXn> }
    { <A1,TY1> , <A2,TY2> , ... , <An,TYn> }
    ```

    respectively, where (a) $n \geq 0$ and (b) for all *j* ($j = 1, 2, ..., n$), types *TXj* and *TYj* have most specific common supertype *Tj* and least specific common subtype *Tj′*. Further, let the values denoted by *RX* and *RY* be relations *rx* and *ry*, respectively. Then:

    a. An expression of the form (*RX*) UNION (*RY*), or logical equivalent thereof, shall be supported and shall denote the **union** of *rx* and *ry*. The declared type of that expression shall have heading

       ```
       { <A1,T1> , <A2,T2> , ... , <An,Tn> }
       ```

b. An expression of the form (*RX*) INTERSECT (*RY*), or logical equivalent thereof, shall be supported and shall denote the **intersection** of *rx* and *ry*. The declared type of that expression shall have heading

```
{ <A1,T1'> , <A2,T2'> , ... , <An,Tn'> }
```

Note: Intersection is a special case of join; given the prescriptions of paragraph d. below, therefore, the present paragraph b. is strictly redundant. We include it for convenience.

c. An expression of the form (*RX*) MINUS (*RY*), or logical equivalent thereof, shall be supported and shall denote the **difference** between *rx* and *ry*, in that order. The declared type of that expression shall have heading

```
{ <A1,TX1> , <A2,TX2> , ... , <An,TXn> }
```

Now let the declared types of relational expressions *RX* and *RY* have headings

```
{ <A1,TX1> , <A2,TX2> , ... , <An,TXn> , <B1,TB1> , ... , <Bp,TBp> }

{ <A1,TY1> , <A2,TY2> , ... , <An,TYn> , <C1,TC1> , ... , <Cq,TCq> }
```

where (a) $n \geq 0$, $p \geq 0$, and $q \geq 0$, and (b) for all *j* (*j* = 1, 2, ..., *n*), types *TXj* and *TYj* have least specific common subtype *Tj'*. Further, let the values denoted by *RX* and *RY* be relations *rx* and *ry*, respectively. Then:

d. An expression of the form (*RX*) JOIN (*RY*), or logical equivalent thereof, shall be supported and shall denote the **join** of *rx* and *ry*. The declared type of that expression shall have heading

```
{ <A1,T1'> , <A2,T2'> , ... , <An,Tn'> ,
             <B1,TB1> , ... , <Bp,TBp> , <C1,TC1> , ... , <Cq,TCq> }
```

Note: Intersection is a special case of join; thus, the prescriptions of the present paragraph d. degenerate to those for intersection (see paragraph b. above) in the case where $p = q = 0$.

14. Let *X* be an expression, let *T* be a type, and let *DT*(*X*) and *T* overlap. Then an operator of the form

```
TREAT_AS_T ( X )
```

(or logical equivalent thereof) shall be supported, with semantics as follows: If *v*(*X*) is not of type *T*, then a type error shall occur; otherwise, the declared type of the invocation TREAT_AS_T(*X*) shall be *T*, and the result of that invocation, *r* say, shall be equal to *v*(*X*) (hence, *MST*(*r*) shall be equal to *MST*(*X*) also).

15. Let *X* be an expression, let *T* be a type, and let *DT*(*X*) and *T* overlap. Then an operator of the form

```
IS_T ( X )
```

(or logical equivalent thereof) shall be supported. The operator shall return TRUE if *v*(*X*) is of type *T*, FALSE otherwise.

16. Let *Op* be a read-only operator, let *P* be a parameter to *Op*, and let *T* be the declared type of *P*. Then the declared type of the argument expression (and therefore, necessarily, the most specific type of the argument as such) corresponding to *P* in an invocation of *Op* shall be allowed to be **any subtype** *T'* of *T*. In other words, the read-only operator *Op* applies to values of type *T* and therefore, necessarily, to values of type *T'*—*The Principle of* ***Read-Only Operator Inheritance***. It follows that such operators are *polymorphic*, since they apply to values of several different types—*The Principle of* ***Read-Only Operator Polymorphism***. It further follows that wherever a value of type *T* is permitted, a value of any subtype of *T* shall also be permitted—*The Principle of* ***Value Substitutability***.

17. Let *Op* be an operator. Then *Op* shall have a *specification signature* and a set of *invocation signatures*. Let the parameters of *Op* and the argument expressions involved in any given invocation of *Op* each constitute an ordered list of *n* elements ($n \geq 0$), such that the *j*th argument expression corresponds to the *j*th parameter ($j = 1, 2, ..., n$). Further, let *PDT* = <*DT1*, *DT2*, ..., *DTn*> be the declared types, in sequence, of those *n* parameters, and let *PDT'* = <*DT1'*, *DT2'*, ..., *DTn'*> be a sequence of types such that *DTj'* is a nonempty subtype of *DTj* ($j = 1, 2, ..., n$). Then:

    a. If *Op* is a read-only operator, the **specification signature** shall consist of the operator name, the sequence *PDT*, and a type (the **declared type** *DT*(*Op*) for, or of, operator *Op*). Also, for each possible sequence *PDT'*, let *OpI* be an invocation of *Op* with argument expressions of declared types as specified by *PDT'*; then there shall exist an **invocation signature** for *OpI*, consisting of that sequence *PDT'* and a type (the **declared type** *DT*(*OpI*) for, or of, invocation *OpI*). *DT*(*OpI*) shall be a subtype of *DT*(*Op*), and the type of the result of *OpI* shall be a subtype of *DT*(*OpI*).

    b. If *Op* is an update operator, the **specification signature** shall consist of the operator name, the sequence *PDT*, and an indication as to which parameters are subject to update. Also, let the sequence *PDT′* be such that an invocation *OpI* of *Op* with argument expressions of declared types as specified by *PDT′* is legitimate (see IM Prescription 19). For each such sequence *PDT′*, there shall exist an **invocation signature** consisting of that sequence *PDT′*.

If two distinct operators (either both read-only or both update operators) have the same name and the same number *n* of parameters, then for some *j* ($1 \le j \le n$) the declared types of their *j*th parameters, as given by their respective specification signatures, shall be disjoint.

Note: Ordered lists or sequences are used in the text of this prescription purely as a convenient basis for defining the various correspondences (e.g., between parameters and their declared types) that the prescription requires. They are not an intrinsic part of the prescription as such. Rather, the implementation is free to establish those correspondences by whatever means it deems suitable, just so long as the overall effect is functionally equivalent to that defined by the foregoing text.

18. Let *Op* be an update operator and let *P* be a parameter to *Op* that is not subject to update. Then *Op* shall behave as a read-only operator as far as *P* is concerned, and all relevant aspects of IM Prescription 16 shall apply, mutatis mutandis.

19. Let *Op* be an update operator, let *P* be a parameter to *Op* that is subject to update, and let *T* be the declared type of *P*. Then it might or might not be the case that the declared type of the argument expression (and therefore, necessarily, the most specific type of the argument as such) corresponding to *P* in an invocation of *Op* shall be allowed to be some proper subtype *T′* of type *T*. It follows that for each such update operator *Op* and for each parameter *P* to *Op* that is subject to update, it shall be necessary to state explicitly for which proper subtypes *T′* of the declared type *T* of parameter *P* operator *Op* shall be inherited—*The Principle of* ***Update Operator Inheritance***. (And if update operator *Op* is not inherited in this way by type *T′*, it shall not be inherited by any proper subtype of type *T′* either.) Update operators shall thus be only conditionally polymorphic—*The Principle of* ***Update Operator Polymorphism***. If *Op* is an update operator and *P* is a parameter to *Op* that is subject to update and *T′* is a proper subtype of the declared type *T* of *P* for which *Op* is inherited, then by definition it shall be possible to invoke *Op* with an argument expression corresponding to parameter *P* that is of declared type *T′*—*The Principle of* ***Variable Substitutability***.

20. Type *T* shall be a **union type** if and only if it is a scalar type and there exists no value that is of type *T* and not of some immediate subtype of *T* (i.e., there exists no value *v* such that *MST*(*v*) is *T*). Moreover:

    a. A type shall be a **dummy type** if and only if either of the following is true:

1. It is one of the types *alpha* and *omega* (see below).

2. It is a union type, has no declared possible representation (and hence no selector), and no regular supertype. *Note:* Type *alpha* in fact satisfies all three of these conditions; type *omega* satisfies the first two only.

A type shall be a **regular type** if and only if it is a scalar type and not a dummy type.

b. Conceptually, there shall be a system defined scalar type called *alpha*, the **maximal type** with respect to every scalar type. That type shall have all of the following properties:

1. It shall contain all scalar values.

2. It shall have no immediate supertypes.

3. It shall be an immediate supertype for every scalar root type in the given set of available types *GSAT*.

No other scalar type shall have any of these properties.

c. Conceptually, there shall be a system defined scalar type called *omega*, the **minimal type** with respect to every scalar type. That type shall have all of the following properties:

1. It shall contain no values at all. (It follows that, as RM Prescription 1 in fact states, it shall have no example value in particular.)

2. It shall have no immediate subtypes.

3. It shall be an immediate subtype for every scalar leaf type in the given set of available types *GSAT*.

No other scalar type shall have any of these properties.

d. The given set of available types *GSAT* shall contain at least one regular scalar type *T* such that *T* is neither a subtype nor a supertype of the required (and system defined) scalar type **boolean**.

21. Type *T* shall be an **empty type** if and only if it is either an empty scalar type or an empty tuple type. Scalar type *T* shall be empty if and only if *T* is type *omega*. Tuple type *T* shall be empty if and only if *T* has at least one attribute that is of some empty type. An empty type shall be permitted as the type of (a) an attribute of a tuple type or relation type; (b) nothing else.

22. Let *T* and *T′* be both tuple types or both relation types. Then type *T′* shall be a **subtype** of type *T*, and type *T* shall be a **supertype** of type *T′*, if and only if (a) *T* and *T′* have the same attribute names *A1*, *A2*, ..., *An* and (b) for all *j* (*j* = 1, 2, ..., *n*), the type of attribute *Aj* of *T′* is a subtype of the type of attribute *Aj* of *T*. Tuple *t* shall be of tuple type *T* if and only if *t* has a heading that is that of some subtype of *T*. Relation *r* shall be of relation type *T* if and only if *r* has a heading that is that of some subtype of *T* (in which case every tuple in the body of *r* shall also have a heading that is that of some subtype of *T*).

23. Let *T* and *T′* be both tuple types or both relation types, with headings

```
{ <A1,T1>  , <A2,T2>  , ... , <An,Tn>  }
{ <A1,T1'> , <A2,T2'> , ... , <An,Tn'> }
```

respectively. Then *T′* shall be a **proper** subtype of *T*, and *T* shall be a **proper** supertype of *T′*, if and only if (a) for all *j* (*j* = 1, 2, ..., *n*), type *Tj′* is a subtype of *Tj* and (b) there exists at least one *j* (*j* = 1, 2, ..., *n*) such that *Tj′* is a proper subtype of *Tj*. Also, *T′* shall be an **immediate** subtype of *T*, and *T* shall be an **immediate** supertype of *T′*, if and only if (a) there exists some *j* (*j* = 1, 2, ..., *n*) such that *Tj′* is an immediate subtype of *Tj* and (b) for all *k* (*k* = 1, 2, ..., *n*, *k* ≠ *j*), *Tk′* = *Tk*. If and only if *T′* is a proper but not an immediate subtype of *T*, then *T′* shall be a **nonimmediate** subtype of *T* and *T* shall be a **nonimmediate** supertype of *T′*.

24. Let *T1*, *T2*, ..., *Tm* (*m* ≥ 0), *T*, and *T′* be all tuple types or all relation types, with headings

```
{ <A1,T11>  , <A2,T12>  , ... , <An,T1n>  }
{ <A1,T21>  , <A2,T22>  , ... , <An,T2n>  }
  .....................................
{ <A1,Tm1>  , <A2,Tm2>  , ... , <An,Tmn>  }
{ <A1,T01>  , <A2,T02>  , ... , <An,T0n>  }
{ <A1,T01'> , <A2,T02'> , ... , <An,T0n'> }
```

respectively. Then:

a. Type *T* shall be a **common supertype** for, or of, types *T1*, *T2*, ..., *Tm* if and only if, for all *j* (*j* = 1, 2, ..., *n*), type *T0j* is a common supertype for types *T1j*, *T2j*, ..., *Tmj*. Further, that type *T* shall be the **most specific** common supertype for *T1*, *T2*, ..., *Tm* if and only if no proper subtype of *T* is also a common supertype for those types.

b. Type *T'* shall be a **common subtype** for, or of, types *T1*, *T2*, ..., *Tm* if and only if, for all *j* (*j* = 1, 2, ..., *n*), type *T0j'* is a common subtype for types *T1j*, *T2j*, ..., *Tmj*. Further, that type *T'* shall be the **least specific** common subtype—also known as the **intersection type** or **intersection subtype**—for *T1*, *T2*, ..., *Tm* if and only if no proper supertype of *T'* is also a common subtype for those types.

Given types *T1*, *T2*, ..., *Tm* as defined above, it can be shown (thanks in particular to IM Prescription 25) that a unique most specific common supertype *T* and a unique least specific common subtype *T'* always exist. In the case of that particular common subtype *T'*, moreover, it is required that whenever a given value is of each of types *T1*, *T2*, ..., *Tm*, it is also of type *T'* (hence the alternative term *intersection type*)—in which case, for all *j* (*j* = 1, 2, ..., *n*), type *T0j'* is the intersection type for types *T1j*, *T2j*, ..., *Tmj*. And it can be shown as a consequence that every tuple value and every relation value has both a unique least specific type and a unique most specific type (regarding the latter, see also IM Prescription 27).

25. Let *T*, *T_alpha*, and *T_omega* be all tuple types or all relation types, with headings

```
{ <A1,T1>       , <A2,T2>       , ... , <An,Tn>       }
{ <A1,T1_alpha> , <A2,T2_alpha> , ... , <An,Tn_alpha> }
{ <A1,T1_omega> , <A2,T2_omega> , ... , <An,Tn_omega> }
```

respectively. Then (a) type *T_alpha* shall be the **maximal type with respect to type *T*** if and only if, for all *j* (*j* = 1, 2, ..., *n*), type *Tj_alpha* is the maximal type with respect to type *Tj*; (b) type *T_omega* shall be the **minimal type with respect to type *T*** if and only if, for all *j* (*j* = 1, 2, ..., *n*), type *Tj_omega* is the minimal type with respect to type *Tj*.

26. A **root type** shall be a scalar root type (see IM Prescription 6), a tuple root type, or a relation root type. A type shall be a **tuple** root type if and only if it is a tuple type *TT* such that every attribute of *TT* is of a root type. A type shall be a **relation** root type if and only if it is a relation type *RT* such that every attribute of *RT* is of a root type.

A **leaf type** shall be a scalar leaf type (see IM Prescription 6), a tuple leaf type, or a relation leaf type. A type shall be a **tuple** leaf type if and only if it is a tuple type *TT* such that every attribute of *TT* is of a leaf type. A type shall be a **relation** leaf type if and only if it is a relation type *RT* such that every attribute of *RT* is of a leaf type.

A **superroot type** shall be a scalar superroot type, a tuple superroot type, or a relation superroot type. A type shall be a **scalar** superroot type if and only if it is type *alpha*. A type *TT* shall be a **tuple** superroot type if and only if it is a proper supertype of some tuple root type (in which case at least one attribute of *TT* must be of some superroot type). A type *RT* shall be a **relation** superroot type if and only if it is a proper supertype of some relation root type (in which case at least one attribute of *RT* must be of some superroot type).

A **subleaf type** shall be a scalar subleaf type, a tuple subleaf type, or a relation subleaf type. A type shall be a **scalar** subleaf type if and only if it is type *omega*. A type *TT* shall be a **tuple** subleaf type if and only if it is a proper subtype of some tuple leaf type (in which case at least one attribute of *TT* must be of some subleaf type). A type *RT* shall be a **relation** subleaf type if and only if it is a proper subtype of some relation leaf type (in which case at least one attribute of *RT* must be of some subleaf type).

27. Let *H* be a heading defined as follows:

```
{ <A1,T1> , <A2,T2> , ... , <An,Tn> }
```

Then:

a. If *t* is a tuple of type TUPLE *H*, meaning *t* shall take the form

```
TUPLE { <A1,MST1,v1> , <A2,MST2,v2> , ... , <An,MSTn,vn> }
```

where, for all *j* (*j* = 1, 2, ..., *n*), type *MSTj* is a subtype of type *Tj* and is the most specific type of value *vj*, then the **most specific** type of *t* shall be

```
TUPLE { <A1,MST1> , <A2,MST2> , ... , <An,MSTn> }
```

b. If *r* is a relation of type RELATION *H*, let the body of *r* consist of tuples *t1*, *t2*, ..., *tm* (*m* ≥ 0). Tuple *ti* (*i* = 1, 2, ..., *m*) shall take the form

```
TUPLE { <A1,MSTi1,vi1> , <A2,MSTi2,vi2> , ... , <An,MSTin,vin> }
```

where, for all *j* (*j* = 1, 2, ..., *n*), type *MSTij* is a subtype of type *Tj* and is the most specific type of value *vij* (note that *MSTij* is different for different tuples *ti*, in general). Then the **most specific** type of *r* shall be

```
RELATION { <A1,MST1> , <A2,MST2> , ... , <An,MSTn> }
```

where, for all *j* (*j* = 1, 2, ..., *n*), type *MSTj* is the most specific common supertype of those most specific types *MSTij*, taken over all tuples *ti*.

28. Let *V* be a tuple variable or relation variable of declared type *T*, and let *T* have attributes *A1*, *A2*, ..., *An*. Then we can model *V* as a named set of named ordered triples of the form <*DTj*,*MSTj*,*vj*> (*j* = 1, 2, ..., *n*), where:

    a. The name of the set is the name of the variable, *V*.

    b. The name of each triple is the name of the corresponding attribute.

    c. *DTj* is the name of the declared type of attribute *Aj*.

    d. *MSTj* is the name of the **most specific type**—also known as the **current** most specific type—for, or of, attribute *Aj*. (If *V* is a relation variable, then the most specific type of *Aj* is the most specific common supertype of the most specific types of the *m* values in *vj*—see the explanation of *vj* below.)

    e. If *V* is a tuple variable, *vj* is a value of most specific type *MSTj*—the **current value** for, or of, attribute *Aj*. If *V* is a relation variable, then let the body of the current value of *V* consist of *m* tuples (*m* ≥ 0); label those tuples (in some arbitrary sequence) “tuple 1,” “tuple 2,” ..., “tuple *m*”; then *vj* is a sequence of *m* values (not necessarily all distinct), being the *Aj* values from tuple 1, tuple 2, ..., tuple *m* (in that order). Note that those *Aj* values are all of type *MSTj*.

We use the notation *DT*(*Aj*), *MST*(*Aj*), *v*(*Aj*) to refer to the *DTj*, *MSTj*, *vj* components, respectively, of attribute *Aj* of this model of tuple variable or relation variable *V*. We also use the notation *DT*(*V*), *MST*(*V*), *v*(*V*) to refer to the overall declared type, overall current most specific type, and overall current value, respectively, of this model of tuple variable or relation variable *V*.

Now let *X* be a tuple expression or relation expression. By definition, *X* specifies an invocation of some tuple operator or relation operator *Op*. Thus, the notation *DTj*(*V*), *MSTj*(*V*), *vj*(*V*) just introduced can be extended in an obvious way to refer to the declared type *DTj*(*X*), the current most specific type *MSTj*(*X*), and the current value *vj*(*X*), respectively, of the *DTj*, *MSTj*, *vj* components, respectively, of attribute *Aj* of tuple expression or relation expression *X*—where *DTj*(*X*) is the declared type of *Aj* for the invocation of *Op* in question (see IM Prescription 17) and is known at compile time, and *MSTj*(*X*) and *vj*(*X*) refer to the result of evaluating *X* and are therefore not known until run time (in general).