Chapter 19

The Inheritance Model

Ruinous inheritance

—Gaius: *The Institutes*

This chapter provides a precise and succinct definition of our model of type inheritance. It consists of a heavily revised version of Chapter 13 from our book *Databases, Types, and the Relational Model: The Third Manifesto,* 3rd edition, Addison-Wesley, 2006 ("the *Manifesto* book" for short). Like that chapter, it mostly just states the various *Inheritance Model Prescriptions* (*IM Prescriptions*) that go to make up that model; in other words, it gives very little by way of discussion or further explanation. (It does give some, though—more than would be required if we were aiming at nothing more than an abstract definition.) *Note:* In most respects, our inheritance model is essentially just a logical consequence of our type theory (and that theory in turn is defined in *The Third Manifesto* itself). It follows that support for *The Third Manifesto,* if it's to be complete, must necessarily include support for the inheritance model in particular.

It should be emphasized that there are significant differences between the version of the model defined herein and the version defined in the *Manifesto* book. Reasons for those differences are explained in detail in Chapter 20. Chapter 21 contains a set of proposals, partly but not wholly repeated from the *Manifesto* book, for extending **Tutorial D** to support the model as defined herein.

Terminology: Throughout what follows, we use the symbols *T* and *T′* generically to refer to a pair of types such that *T′* is a subtype of *T* (equivalently, such that *T* is a supertype of *T′*). Keep in mind that types *T* and *T′* aren't limited to being scalar types specifically, barring explicit statements to the contrary. Note too that distinct types have distinct names; in particular, if *T′* is a proper subtype of *T* (see IM Prescription 4), then their names will be distinct, even if that proper subtype *T′* of *T* isn't a proper subset of *T* (see IM Prescription 2). Also, we assume that all of the types under discussion, including the maximal and minimal types discussed in IM Prescriptions 20 and 24, are members of some given set of types *GST;* 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 *GST* (though the only explicit mention of that set is in IM Prescription 20, q.v.).

Wherever there's a discrepancy between the present chapter and Chapter 13 of the *Manifesto* book, the present chapter should be taken as superseding.

IM PRESCRIPTIONS

1. *T* and *T′* shall indeed both be types; i.e., each shall be a named, finite 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*). *Note:* In the case of scalar types, at least, we would normally expect proper subtypes to be proper subsets (see IM Prescription 4); in other words, we would normally expect there to exist, so long as *T* and *T′* are distinct, at least one value of type *T* that is not of type *T′*. Certain of the prescriptions that follow have been designed on the basis of this expectation; however, they do not formally depend on it.

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

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

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

6. If and only if T' is a proper subtype of T and there is no type that is both a proper supertype of T' and a proper subtype of T, then T' shall be an **immediate** subtype of T, and T shall be an **immediate** supertype of T'. A type that has some maximal type—see IM Prescriptions 20 and 24—as its sole immediate supertype shall be a **root** type; a type that has some minimal type—again, see IM Prescriptions 20 and 24—as its sole immediate subtype shall be a **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 they are the same type or there exists at least one value that is common to both. Distinct root types shall be disjoint.

8. Let $T1, T2, ..., Tm$ ($m \geqslant 0$), 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.

   Note: 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 can also be shown 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 further be shown that every scalar value has both a unique least specific type and a unique most specific type (regarding the latter, 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. Then 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 is of type T and it satisfies constraint SC. *Note:* We would normally expect there to exist at least one value of type T that does not satisfy constraint SC (see IM Prescription 2).

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. Attributes $\langle Ax,DTx\rangle$ of relation rx and $\langle Ay,DTy\rangle$ of relation ry shall **correspond** if and only if their names Ax and Ay are the same, A say, and their declared types DTx and DTy have a common supertype. Then:

    a. It shall be possible to form the **union** of rx and ry if and only if each attribute of rx corresponds to some attribute of ry and vice versa. For each pair of corresponding attributes $\langle A,DTx\rangle$ and $\langle A,DTy\rangle$, the declared type of the corresponding attribute in the result of the union shall be the most specific common supertype of DTx and DTy. *Note:* In practice, the implementation might want to outlaw, or at least flag, any attempt to form such a union if DTx and DTy are not subtypes of the same root type.

    b. It shall be possible to form the **intersection** of rx and ry if and only if each attribute of rx corresponds to some attribute of ry and vice versa. For each pair of corresponding attributes $\langle A,DTx\rangle$ and $\langle A,DTy\rangle$, the declared type of the corresponding attribute in the result of the union shall be the least specific common subtype of DTx and DTy. *Note:* In practice, the implementation might want to outlaw, or at least flag, any attempt to form such an intersection if DTx and DTy are not supertypes of the same leaf type. Also, intersection is a special case of join; given the prescriptions of paragraph d. below, therefore, the present paragraph is strictly redundant. We include it for convenience.

    c. It shall be possible to form the **difference** between rx and ry, in that order, if and only if every attribute of rx corresponds to some attribute of ry and vice versa. For each pair of corresponding attributes $\langle A,DTx\rangle$ and $\langle A,DTy\rangle$, the declared type of the corresponding attribute in the result of the difference shall be DTx. *Note:* In practice, the implementation might want to outlaw, or at least flag, any attempt to form such a difference if DTx and DTy are not subtypes of the same root type, and possibly also if DTx and DTy are not supertypes of the same leaf type.

    d. It shall be possible to form the **join** of rx and ry if and only if no attribute of rx that fails to correspond to an attribute of ry has the same name as any attribute of ry and vice versa. For each pair of corresponding attributes $\langle A,DTx\rangle$ and $\langle A,DTy\rangle$, the declared type of the corresponding attribute in the result of the join shall be the least specific common subtype of DTx and DTy. *Note:* In practice, the implementation might want to outlaw, or at least flag, any attempt to form such a join if DTx and DTy are not supertypes of the same leaf type. Also, intersection is a special case of join; thus, the prescriptions of the present paragraph degenerate to those for intersection in the case where every attribute of *rx* corresponds to some attribute of *ry* and vice versa.

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. We refer to such operators generically as "TREAT" or "TREAT AS" operators; their semantics are as follows. First, if *v*(*X*) is not of type *T,* then a type error shall occur. Otherwise:

    a. If the TREAT invocation appears in a "source" position—in particular, on the right side of an assignment—then the declared type of that invocation shall be *T,* and the invocation shall yield a result, *r* say, with *v*(*r*) equal to *v*(*X*) (and hence *MST*(*r*) equal to *MST*(*X*) also).

    b. If the TREAT invocation appears in a "target" position—in particular, on the left side of an assignment—then that invocation shall act as a pseudovariable reference, which means it shall designate a pseudovariable *X'* with *DT*(*X'*) equal to *T,* *v*(*X'*) equal to *v*(*X*), and *MST*(*X'*) equal to *MST*(*X*).

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 a read-only operator. Then *Op* shall have exactly one **specification signature,** denoting that operator as perceived by potential users. The specification signature for *Op* shall consist of the operator name and a nonempty set of *invocation signatures*. For definiteness, assume the parameters of *Op* and the argument expressions involved in any given invocation of *Op* each constitute an ordered list of *n* elements ($n \geqslant 0$), 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* ($j = 1, 2, ..., n$). For each such sequence *PDT',* there shall exist an **invocation signature** consisting of the operator name and a specification of the declared type of the result of an invocation of *Op* with argument expressions of declared types as specified by *PDT'* (the **declared type** for, or of, such an invocation).

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 is 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 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 types *GST*.

        No other scalar type shall have any of these properties (unless the given set of types *GST* contains just one regular type—necessarily type **boolean**—in which unlikely case that type will of course satisfy the first property).

    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 types *GST*.

        No other scalar type shall have any of these properties.

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 declared 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 some subtype of tuple type *T* if and only if the heading of *t* is that of some subtype of *T*. Relation *r* shall be of some subtype of relation type *T* if and only if the heading of *r* is that of some subtype of *T* (in which case every tuple in the body of *r* shall necessarily also have a heading that is that of some subtype of *T*).

23. 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. Further, for all *j* (*j* = 1, 2, ..., *n*), let types *T1j, T2j, ..., Tmj* have a common subtype (and hence a common supertype also). 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.

Note: Given types *T1, T2, ..., Tm* as defined above, it can be shown (thanks in particular to IM Prescription 24) 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 can also be shown 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 further be shown 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 25).

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

25. Let {*H*} be a heading defined as follows:

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

Then:

a. If *t* is a tuple of type some subtype of TUPLE {*H*}—meaning *t* is of the form

```
{ <A1,T1',v1> , <A2,T2',v2> , ... , <An,Tn',vn> }
```

where, for all *j* (*j* = 1, 2, ..., *n*), type *Tj'* is a subtype of type *Tj* and *vj* is a value of type *Tj'*—then the **most specific** type of *t* shall be

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

where, for all *j* (*j* = 1, 2, ..., *n*), type *MSTj* is the most specific type of value *vj*.

b. If *r* is a relation of type some subtype of RELATION {*H*}—meaning each tuple in the body of *r* can be regarded without loss of generality as being of the form

```
{ <A1,T1',v1> , <A2,T2',v2> , ... , <An,Tn',vn> }
```

where, for all *j* (*j* = 1, 2, ..., *n*), type *Tj'* is a subtype of type *Tj* and is the most specific type of value *vj* (note that distinct tuples in the body of *r* will be of distinct most specific types, in general; thus, type *Tj'* varies over the tuples in the body of *r*)—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 *Tj',* taken over all tuples in the body of *r*.

26. Let *V* be a tuple variable or relation variable of declared type *T,* and let the heading of *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 ($m \geqslant 0$); 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).