Chapter 13

T h e I n h e r i t a n c e M o d e l

Introduction
IM Prescriptions
Recent inheritance model changes

INTRODUCTION

In this chapter, we simply state the various IM Prescriptions that go to make up our model of subtyping and inheritance, with no attempt at discussion or further explanation. The material is provided primarily for reference; you probably should not even attempt to read it straight through (at least, not on a first reading).

A note on terminology: Throughout what follows, we assume that all of the types under discussion are members of some given set of types *S;* in particular, the definitions of the terms *root type* and *leaf type* are to be interpreted in the context of that set *S*. Also, 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′*). You might find it helpful to think of *T* and *T′* as ELLIPSE and CIRCLE, respectively. Keep in mind, however, the fact that, in general, they are *not* limited to being scalar types specifically, barring explicit statements to the contrary.

IM PRESCRIPTIONS

1. *T* and *T′* shall indeed both 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*). Moreover, if *T* and *T′* are distinct (see IM Prescriptions 3 and 4), then there shall exist at least one value of type *T* that is not 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. 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 is not an immediate subtype of any type shall be a **root** type. A type that is not an immediate supertype of any type shall be a **leaf** type.

7. If *T1* and *T2* are distinct root types, then they shall be disjoint; i.e., no value shall be of both type *T1* and type *T2*.

8. Every set of types *T1, T2, ..., Tn* shall have a common subtype *T′* such that a given value is of each of the types *T1, T2, ..., Tn* if and only if it is of type *T′*.

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 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*.

   Now let *X* be a scalar expression. By definition, *X* specifies an invocation of some scalar operator *Op* (where the arguments, if any, to that invocation of *Op* are specified as expressions in turn). 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 *DT*(*Op*) 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 proper supertype (see IM Prescription 20), and let *T′* be an 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*. There shall exist at least one value of type *T* that does not satisfy constraint *SC*.

11. Consider the assignment

    V := *X*

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

12. Consider the equality comparison

    Y = *X*

    (where *Y* and *X* are expressions). *DT*(*X*) and *DT*(*Y*) shall have a nonempty common subtype. The comparison shall return TRUE if *MST*(*Y*) is equal to *MST*(*X*) and *v*(*Y*) is equal to *v*(*X*), FALSE otherwise.

13. Let *rx* and *ry* be relations with a common attribute *A,* and let the declared types of *A* in *rx* and *ry* be *DTx*(*A*) and *DTy*(*A*), respectively. Consider the join of *rx* and *ry* (necessarily over *A,* at least in part). *DTx*(*A*) and *DTy*(*A*) shall have a nonempty common subtype and hence shall also have a most specific common supertype, *T* say. Then the declared type of *A* in the result of the join shall be *T*.

    Analogous remarks apply to union, intersection, and difference operators. That is, in each case:

    a. Corresponding attributes of the operands shall be such that their declared types have a nonempty common subtype.

    b. The declared type of the corresponding attribute of the result shall be the corresponding most specific common supertype.

14. Let *X* be an expression, let *T* be a type, and let *DT*(*X*) and *T* have a nonempty common subtype. 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 *MST*(*X*) is not a subtype of *T,* then a type error shall occur. Otherwise:

a. If the TREAT invocation appears in a "source" position (for example, 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 *MST*(*r*) equal to *MST*(*X*) and *v*(*r*) equal to *v*(*X*).

b. If the TREAT invocation appears in a "target" position (for example, on the left side of an assignment), then that invocation shall act as a pseudovariable reference, which means it shall actually *designate* its argument *X* (more precisely, it shall designate a version of *X* for which *DT*(*X*) is equal to *T* but *MST*(*X*) and *v*(*X*) are unchanged).

15. Let *X* be an expression, let *T* be a type, and let *DT*(*X*) and *T* have a nonempty common subtype. Then a logical 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 (and therefore, necessarily, the most specific type) of the argument *A* 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. Any given operator *Op* shall have exactly one **specification signature,** a nonempty set of **version signatures,** and a nonempty set of **invocation signatures**. For definiteness, assume the parameters of *Op* and the arguments appearing in any given invocation of *Op* each constitute an ordered list of *n* elements ($n \geq 0$), such that the *i*th argument corresponds to the *i*th parameter. Then:

a. The specification signature shall denote *Op* as perceived by potential users. It shall consist of the operator name *Op,* the declared types (in order) of the parameters to *Op,* and the declared type of the result, if any, of executing *Op*.

b. There shall be one version signature for each implementation version *V* of *Op*. Each such signature shall consist of the operator name *Op* (and possibly the version name *V*), the declared types (in order) of the parameters to *V,* and the declared type of the result, if any, of executing *V*.

c. There shall be one invocation signature for each possible combination of most specific argument types to an invocation of *Op*. Each such signature shall consist of the operator name *Op* and the pertinent combination of most specific argument types (in order). *Note:* The invocation signatures for *Op* can easily be derived from the corresponding specification signature, but the concepts are logically distinct.

Every version of *Op* shall implement the same semantics.

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 therefore 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 (and therefore, necessarily, the current most specific type) of the argument *A* corresponding to *P* in an invocation of *Op* shall be allowed to be a proper subtype 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 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 corresponding to parameter *P* that is of declared type *T′*——*The Principle of* ***(Variable) Substitutability***.

20. A **union type** is a type *T* such that 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*). A **dummy type** is a union type that has no declared possible representation (and hence no selector); a given union type shall be permitted to be a dummy type if and only if it is empty or it has no regular immediate supertype (where a **regular type** is a type that is not a dummy type). Moreover:

    a. Conceptually, there is a special scalar dummy type, *alpha,* that contains all scalar values. Type *alpha* is **the maximal type** with respect to every scalar type; by definition, it has no declared possible representation and no immediate supertypes.

    b. Conceptually, there is a special scalar dummy type, *omega,* that contains no values at all. Type *omega* is **the minimal type** with respect to every scalar type; by definition, it has no declared possible representation and no immediate subtypes.

21. Let types *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 type *T'* is a **subtype** of type *T* (equivalently, type *T* is a **supertype** of type *T'*) if and only if, for all *i* (*i* = 1, 2, ..., *n*), type *Ti'* is a subtype of type *Ti* (equivalently, type *Ti* is a supertype of type *Ti'*).

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

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

    Then:

    a. *t* is a tuple that **conforms** to heading {*H*} if and only if *t* is of the form

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

       where, for all *i* (*i* = 1, 2, ..., *n*), type *Ti'* is a subtype of type *Ti* and *vi* is a value of type *Ti'*.

    b. *r* is a relation that **conforms** to heading {*H*} if and only if *r* consists of a heading and a body, where:

- The heading of *r* is of the form

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

  where, for all *i* (*i* = 1, 2, ..., *n*), type *Ti'* is a subtype of type *Ti*.

- The body of *r* is a set of tuples, all of which conform to the heading of *r*.

23. Let types *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* are **the maximal type with respect to type *T*** and **the minimal type with respect to type *T*,** respectively, if and only if, for all *i* (*i* = 1, 2, ..., *n*), type *Ti_alpha* is the maximal type with respect to type *Ti* and type *Ti_omega* is the minimal type with respect to type *Ti*.

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

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

    Then:

    a. If *t* is a tuple that conforms to *H*——meaning *t* is of the form

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

       where, for all *i* (*i* = 1, 2, ..., *n*), type *Ti'* is a subtype of type *Ti* and *vi* is a value of type *Ti'*——then the **most specific** type of *t* is

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

       where, for all *i* (*i* = 1, 2, ..., *n*), type *MSTi* is the most specific type of value *vi*.

    b. If *r* is a relation that conforms to *H*——meaning the body of *r* is a set of tuples, each of which has as its most specific type a type that is a subtype of the type TUPLE{*H*}, and meaning further that each such tuple can be regarded without loss of generality as being of the form

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

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

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

where, for all *i* (*i* = 1, 2, ..., *n*), type *MSTi* is the most specific common supertype of those most specific types *Ti',* taken over all tuples in the body of *r*.

25. Let tuple or relation variable *V* be 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 <*DTi,MSTi,vi*>, where:

    a. The name of the set is the name of the variable (*V* in the example).

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

    c. *DTi* is the name of the declared type of attribute *Ai*.

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

    e. If *V* is a tuple variable, *vi* is a value of most specific type *MSTi*——the **current value** for, or of, attribute *Ai*. If *V* is a relation variable, then let the body of the current value of *V* consist of *m* tuples; label those tuples (in some arbitrary sequence) "tuple 1," "tuple 2," ..., "tuple *m*"; then *vi* is a sequence of *m* values (not necessarily all distinct), being the *Ai* values from tuple 1, tuple 2, ..., tuple *m* (in that order); note that all of those values are of type *MSTi*.

    We use the notation *DT*(*Ai*), *MST*(*Ai*), *v*(*Ai*) to refer to the *DTi, MSTi, vi* components, respectively, of attribute *Ai* of this model of tuple 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 or relation variable *V*.

Now let *X* be a tuple or relation expression. By definition, *X* specifies an invocation of some tuple or relation operator *Op* (where the arguments, if any, to that invocation of *Op* are specified as expressions in turn). Thus, the notation *DTi*(*V*), *MSTi*(*V*), *vi*(*V*) just introduced can be extended in an obvious way to refer to the declared type *DTi*(*X*), the current most specific type *MSTi*(*X*), and the current value *vi*(*X*), respectively, of the *DTi, MSTi, vi* components, respectively, of attribute *Ai* of tuple or relation expression *X*——where *DTi*(*X*) is known at compile time, and *MSTi*(*X*) and *vi*(*X*) refer to the result of evaluating *X* and are therefore not known until run time (in general).

RECENT INHERITANCE MODEL CHANGES

There are a few differences between the inheritance model as defined in the present chapter and the version documented in this book's predecessor (reference [83]). For the benefit of readers who might be familiar with that earlier version, we summarize the main differences here.

- The root and leaf type concepts are no longer regarded as absolutes but are instead defined relative to some given set of types (which can be thought of, informally, as a type hierarchy or type graph). This change primarily affects IM Prescriptions 6 and 7.

- The wording of several prescriptions has been changed to cover (a) both single and multiple inheritance and (b) both scalar and tuple/relation types. These changes——which are not intended to affect the model as such——apply primarily to IM Prescriptions 7 and 8.

- In IM Prescription 12, the requirement that *DT*(*X*) and *DT*(*Y*) shall have a common supertype has been replaced by the stronger requirement that *DT*(*X*) and *DT*(*Y*) shall have a nonempty common subtype. (The fact that the new requirement is stronger than the old one might not be obvious but is proved in Chapter 15.)

- Analogously, in IM Prescription 13, the requirement that *DTx*(*A*) and *DTy*(*A*) shall have a common supertype has been replaced by the stronger requirement that *DTx*(*A*) and *DTy*(*A*) shall have a nonempty common subtype.

- In IM Prescription 14, TREAT DOWN has been renamed TREAT AS or (more usually) just TREAT.

- IM Prescriptions 14 and 15 have been generalized slightly (previously we required *T* to be a subtype of *DT*(*X*); now we require only that they have a nonempty common subtype).

- IM Prescription 20 has been corrected slightly (previously it required a dummy type to have no regular immediate supertypes, but this rule overlooked the possibility that a dummy type might be empty).

In addition to all of the foregoing, almost all of the prescriptions have been reworded somewhat. However, those revisions in themselves are not intended to induce any changes in what is being described.

***** End of Chapter 13 *****

