Draft of 17 February 2003

Proving Pointer Program Properties

Part 3: Considering individual fields

Bertrand Meyer

ABSTRACT

In object-oriented programming, the overall reference structure is the combination of individual structures induced by the attributes of the underlying classes. This article makes the transition from a coarse-grained view to one allowing individual consideration of object fields.

This is part of a series of articles. See here for links to the others.

The previous article in this series has presented a coarse-grain model of the run-time object structure — the Relation Model — based on a single relation *attached*. It is time now to refine the model so that it will more directly reflect the properties of object structures in an object-oriented language.

3.1 MODELING ATTRIBUTES

Let's examine what kind of relation *attached* represents in practice. Consider a class, with its attributes only, routines omitted

```
class PERSON feature
        age: INTEGER
        landlord, loved, servant: PERSON
        name: STRING
end
```

assuming similar descriptions are available for *STRING* and any other needed classes. The attribute *age* denotes the kind of expanded field that we ignore at this stage of the discussion. Class *STRING* denotes strings; we assume that, as in Eiffel, strings are objects, so that *name* denotes a reference field.

Given any instance *p* of class *PERSON* the relation *attached* will contain exactly four pairs having *p* as their first element: one each for *landlord*, *loved*, *servant* and *name*.

> The figure on page 4 of the previous article omitted the field *age* and treated *STRING* as if it were an expanded rather than reference type, leaving only the fields *landlord*, *loved*, *servant*. The pairs for the top-left object, numbered 8, were: [8, 8], [8, 9], [8, 12].

Here we can say that all instances of class *PERSON* have a "fan-out" of four, where the fan-out of an object *o* is the number of pairs having *o* as their first element. In a typed object-oriented language, this is a general property: the fan-out is the same for all instances of a class. In addition, the targets of the links are always of the same types: here *landlord*, *loved* and *servant* will always lead (if not void) to instances of *PERSON*, and *name* to instances of *STRING*. This is an important property of the corresponding run-time object structure, to which we may give a name:

Class-Based Nature of Objects

In a typed object-oriented language as considered in this discussion, every class defines a set of links and link types applicable to all its instances.

In the Relation Model, *attached* is an arbitrary relation; this doesn't address the Class-Based Nature of Objects. To have a realistic model and prove all relevant properties, we must take this property into consideration.

A first attempt

Let us first briefly examine an approach that has been used by several authors and see why it is not appropriate.

That approach, which we may call the Attribute Function Model, probably seems the most natural: it refines the Relation Model by considering that each class defines a two-argument function

$$attribute\_link: Attributes \times Addresses \nrightarrow Addresses$$

where $\times$ is cartesian product, *Attributes* is the set of possible attribute names (such as *landlord* etc. in the example), and $A \nrightarrow B$ is the set of functions, possibly partial, from A to B. (As before we are only interested in the set of finite functions, $A \twoheadrightarrow B$, but this makes no difference since *Addresses* is finite.) So for an instance p of *PERSON* the function *attribute_link* will define the following values, and these only:

$$\begin{array}{l} attribute\_link\ (\text{"}PERSON.landlord\text{"}, p) \\ attribute\_link\ (\text{"}PERSON.loved\text{"}, p) \\ attribute\_link\ (\text{"}PERSON.servant\text{"}, p) \\ attribute\_link\ (\text{"}PERSON.name\text{"}, p) \end{array}$$

using attribute names of the form *CLASS_NAME*.*attribute_name* to identify each attribute uniquely. (More later on this convention.)

→"The Function Union Model", , page 4.

The relationship to the original Relation Model is easy to define. If we define the function *link*, with the signature

$$link: Attributes \nrightarrow (Addresses \nrightarrow Addresses)$$

as the specialization ("currying") of *attribute_link* on its first argument, that is to say, $link\ (a)$, for any attribute a, is the function $link_a$ such that, for any address *addr* for which it is defined

$$link_a\ (addr) = attribute\_link\ (a, addr)$$

then the relation *attached* of the Relation Model is in direct correspondence with the function *link*:

[T51] $attached = \bigcup_{a : Attributes} link\ (a)$

However intuitive at first, this Attribute Function Model is impractical for our general goal of proving properties of sophisticated programs and libraries. The reason is simply that the function *attribute_link* takes two arguments. This makes it hard to derive properties of complex structures by combining properties of simpler ones — our only hope for tackling complexity.

If we stick to **one-argument functions** f and g we will be able to deduce, from properties of f and g, properties of their union $f \cup g$, their intersection $f \cap g$, their closures f^* and g^* — which tell us which objects can be reached from a given object — and various combinations of these operations. By applying these rules repeatedly we can deduce properties of the entire structure. In other words, a model with one-argument functions will scale up. With the Attribute Function model we have to drag along the function *attribute_link*: $Attributes \times Addresses \nrightarrow Addresses$ which treats its two arguments — attribute tag and object — symmetrically although they have different roles. Union and intersection don't represent any relevant property, and there is no transitive closure. This means we can't use the strategy of first studying properties of various attributes in isolation, then combining them through basic set operations. The model doesn't scale.

The Function Union Model

To enjoy the convenience of one-argument functions we will simply consider each attribute of a class, separately, as defining a function on the object structure. Then the relation *attached* of the Relation Model will be the union of all such functions.

In our example class declaration

```
class PERSON feature
    age: INTEGER
    landlord, loved, servant: PERSON
    name: STRING
end
```

each attribute, such as *landlord* or *name*, defines at run time a function from objects to objects; for *name*, the function's domain is the set of instances of *PERSON*, and its range is a subset of the set of instances of *STRING*.

For any attribute *a*, we call the corresponding function $link_a$ as above:

$$link_a : Addresses \nrightarrow Addresses$$

This function represents all the inter-object links induced by the attribute *a*.

To avoid referring explicitly to classes throughout, we consider that an attribute name *a* — a member of *Attributes* — represents "a certain attribute of a certain class"; so *a* carries within itself the indication of its class. This was achieved by the notation previously used to denote attributes: *CLASS_NAME*.*attribute_name*, for example *PERSON*.*landlord*. ← *Page 3.*

That two attributes from two unrelated classes may have the same name doesn't affect this property: they are still different members of the set *Attributes*.

With inheritance, an attribute of a class may also be available in another class. For example, if class *PERSON* has a descendant *CHILD*, and we apply *loved* to an instance of a *CHILD*, we are using the same member *PERSON*.*loved* of *Attributes*. Here there is no *CHILD*.*loved*. Note that the name of the attribute in the class is once again irrelevant: in Eiffel, which clearly distinguishes between *features* and *feature names*, the attribute *loved* may have a different name in *CHILD*, but it's still the attribute we are calling *CHILD*.*loved*; conversely (and perversely) *CHILD* could have another attribute called *loved* bearing no relation to the original. At the semantic level, where we are dealing, such naming and other source language issues must have been previously resolved.

The relation *attached* at the center of our model is the union of all $link_a$ functions, for all attributes *a* of all classes:

Function Union property

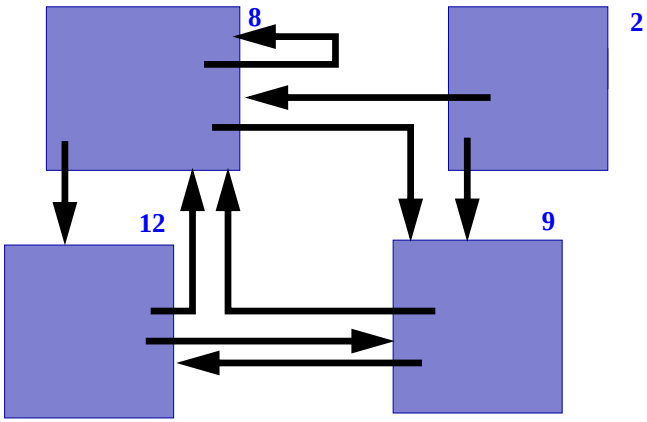
[T52] $$attached = \bigcup_{a\,:\,Attributes} link_a$$

where $link_a$ is a function, denoting the links induced by an attribute *a* of a class.

← *This is the same property as [T51].*

The Function Union property describes *attached* as a union of functions. It gives its name to the model's new refinement: the **Function Union model**.

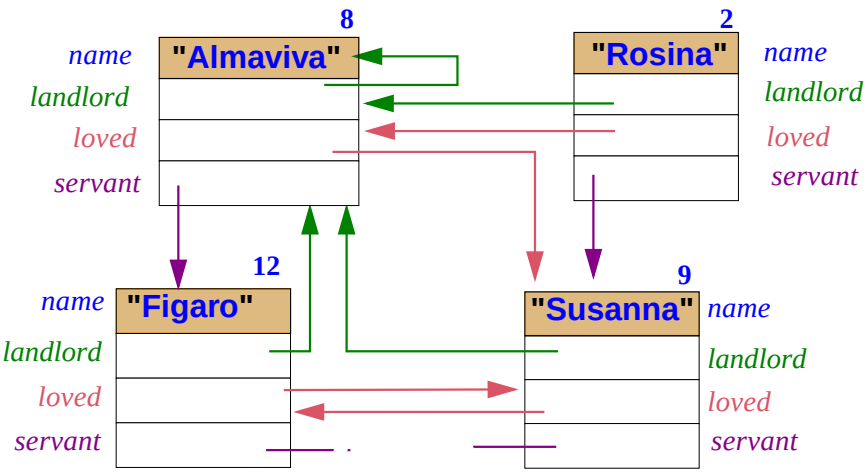
We may interpret the Function Union property visually by looking again at the figure that showed our example structure in its relational form, illustrating the relation *attached*:



References collapsed into a relation*: *attached

(*First shown on page 4 of part 2.*)

and recalling that it was only intended as a coarse-grain view of the functional form, which in its fine-grain form shows individual links:

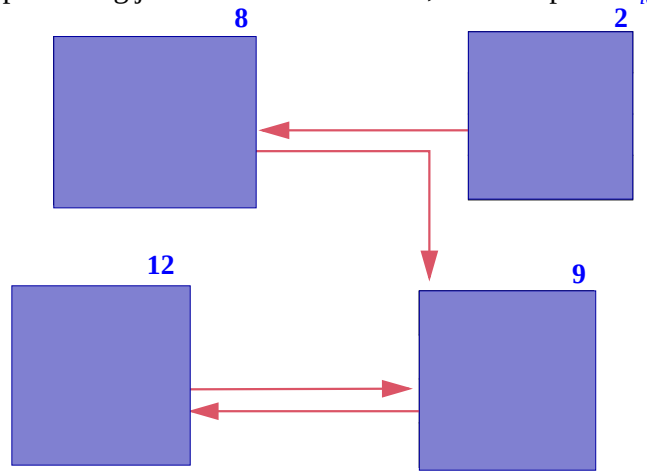


From an object store

(*First shown on page 4 of part 2.*)

For simplicity, as noted, the figure shows name fields as if they were expanded, although they are actually references to STRING objects. It also ignores age fields. This does not affect the discussion.

The Function Union model considers that each attribute represents a mathematical function; here it would call these functions $link_{landlord}$, $link_{loved}$ and so on. Each is identified in the figure by a different color. We may draw the graph representing just one of the functions, for example $link_{loved}$:



A single attribute function: $link_{loved}$

The *attached* graph is the superimposition of all such individual attribute graphs. With the Function Union model we study each attribute graph separately, and define *attached* as their union.

[T51] gives the connection with the previous, discarded model — the Attribute Function model. The two models are indeed conceptually equivalent, but we get a considerable gain of convenience by switching from a function in $Attributes \times Addresses \nrightarrow Addresses$ to one-argument functions from *Addresses* to itself.

In manipulating functions, one should be careful of the plain union operator because the union of two functions is a relation but not, in the general case, a function. In [T52] we don't mind, since the left-hand side, *attached*, was defined from the start as a relation.

More precisely, a union of functions is a function only if they have disjoint domains, or at least coincide on their domains' intersection. This is not necessarily the case here since $link_a$ functions have the same domain for attributes coming from the same class. For an instance *o* of a class having attributes *a* and *b*, $link_a$ and $link_b$ yield the same value if the corresponding reference fields link *o* to the same objects. In our example graph object **2**, *Rosina*, links to object **8**, *Almaviva*, through both *loved* and *landlord*.

The use of functions in the Function Union Model makes it a good intuitive picture of the practice of object-oriented programming. For any object *o* of type *C* and any attribute *a* of the class, there is exactly one value — denoted in most languages by *o*.*a* — for the corresponding field. (For a non-expanded attribute that value is either *Void* or a reference to another object, which makes no difference in our model.) This is the $link_a$ function.

A strategy

Our newest model suggests a two-step strategy for studying object structures:

1 • First, focus on each relevant attribute separately, studying the properties of the object subgraph that it induces. This leads quickly to interesting properties.

2 • Then, by applying the Union operator, infer properties of the full relation *attached*, when needed, from properties of the individual $link_a$ functions that make up that relation.

We will be particularly interested, for step 2, in properties that are preserved by the union operator. We will call them **union-stable**. Here is a first repertoire of union-stability theorems, with straightforward proofs:

[T53] $f(\bullet A \bullet) \cup g(\bullet A \bullet) = (f \cup g)(\bullet A \bullet)$
[T54] $f^* \cup g^* \subseteq (f \cup g)^*$
[T55] $f^+ \cup g^+ \subseteq (f \cup g)^+$

Part 4 applies the strategy just defined to modeling the object structures induced by classes describing fundamental data structures, and proving their run-time properties.