Information Modeling and Higher-order Types

Terry Halpin

Northface University
Salt Lake City, Utah, USA.
e-mail: terry.halpin@northface.edu

Abstract: While some information modeling approaches (e.g. the Relational Model, and Object-Role Modeling) are typically formalized using first-order logic, other approaches to information modeling include support for higher-order types. There appear to be three main reasons for requiring higher-order types: (1) to permit instances of categorization types to be types themselves (e.g. the Unified Modeling Language introduced power types for this purpose); (2) to directly support quantification over sets and general concepts; (3) to specify business rules that cross levels/metalevels (or ignore level distinctions) in the same model. As the move to higher-order logic may add considerable complexity to the task of formalizing and implementing a modeling approach, it is worth investigating whether the same practical modeling objectives can be met while staying within a first-order framework. This paper examines some key issues involved, suggests techniques for retaining a first-order formalization, and also makes some suggestions for adopting a higher-order semantics.

1 Introduction

Following Codd's use of first-order logic to formally underpin the relational model of data [4], most formalizations of information modeling approaches restricted their logical foundations to first-order (where quantification is permitted over individuals only, not predicates). This is the case for Entity Relationship (ER) modeling [3], as well as Object-Role Modeling (ORM) and its variants [e.g. 2, 12, 13]. A full formalization of ORM's fact-oriented (attribute-free) approach to information modeling was first provided in [11], with alternative formalizations supplied later [5, 15]. In contrast, the Unified Modeling Language (UML) [19, 20, 22] introduced the notion of powertypes, whose instances may themselves be types, thus requiring higher-order semantics.

There appear to be three main arguments for requiring higher-order types to logically underpin information modeling semantics:

- to allow one to think of instances of certain categorization types (e.g. AccountType, CarModel) as being types themselves (as for UML powertypes);
- to formalize very directly the semantics of flexible data structures where attribute entries may themselves denote sets or general concepts (e.g. object-relational tables in non-first normal form);
- to allow one to specify business rules that seem to cross levels/metalevels (or ignore level distinctions) in the same model (e.g. the Finance department is responsible for defining the possible values of AccountType).

As the move to higher-order logic may add considerable complexity to the task of formalizing and implementing a modeling approach, it is worth investigating whether the same practical modeling objectives can be achieved while staying within a first-order framework. This paper examines some key issues involved, suggests techniques to maintain a first-order formalization, and also makes some suggestions for adopting a higher-order semantics. The examples are presented in ORM and/or UML notation, but the issues are relevant to all information modeling approaches.

Section 2 addresses the question of whether instances of categorization types may themselves be types. Section 3 considers what higher-order logic may be appropriate to cater for higher-order types. Section 4 provides an alternate first-order approach to categorization schemes. Section 5 discusses whether higher-order logic is needed to formalize the presence of set-structures or the ability to cross levels/meta-levels in the same model. Section 6 summarizes the main results, suggests topics for future research, and lists references for further reading.

2 May instances of categorization-types be types themselves?

Figure 1 depicts a simple schema in (a) ORM and (b) UML notation. The ORM diagram may be interpreted as follows. Four object types (Account, AccountType, SavingsAccount, and InterestRate) are depicted as named ellipses. Here "Account" means bank account. The thick arrow indicates that SavingsAccount is a subtype of Account. As in logic, a predicate is a proposition with object-holes in it. In ORM, a predicate is treated as a named sequence of one or more roles, each of which is depicted as a box. Combining a predicate with its sequence of object types produces a fact type (e.g. Account is of AccountType, SavingsAccount earns InterestRate). Simple identification schemes may be abbreviated in parentheses. For example, Account(Nr) abbreviates the injective (1:1 into) fact type Account has AccountNr.

The value constraint {'CheckingAccount', 'SavingsAccount'} indicates the possible names of account types. Arrow-tipped lines across one or more roles denote uniqueness constraints, indicating that instantiations of that role sequence must be unique. For example, the uniqueness constraint on the first role of Account is of AccountType indicates that entries in the fact column for that role must be unique, which may be formally verbalized as: **each** Account is of **at most one** AccountType.

A solid dot (possibly circled) connected to a set of one or more roles denotes a mandatory constraint over that role set. For example, the mandatory dot connected to the first role of Account is of AccountType indicates that **each** Account is of **some** AccountType. The text beneath the diagram provides a formal definition for the SavingsAccount subtype. For discussion purposes, the two AccountType instances named "CheckingAccount" and "SavingsAccount", are depicted here as a white dot and shaded dot respectively.

In the UML class diagram, AccountType is an enumerated type with two values[1], and provides the type for the accountType attribute of Account, while SavingsAccount is a subclass of the Account class.

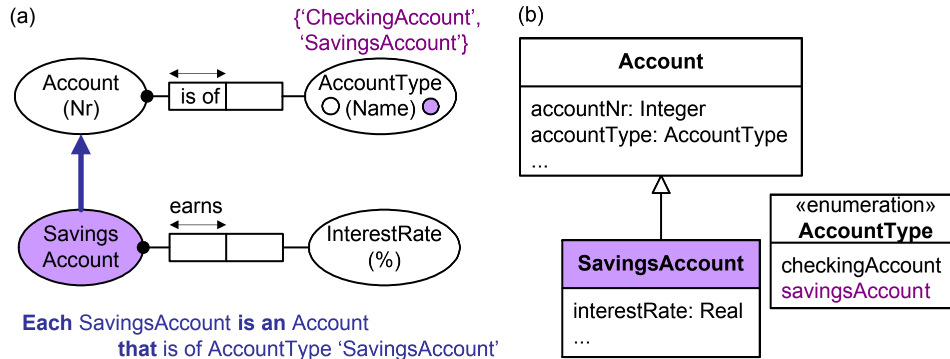

Figure 1 Is the AccountType instance for "SavingsAccount" identical to the subtype SavingsAccount?

In practice, we would normally remove or expand the value constraint, to allow other types of account (e.g. LoanAccount) without modifying the schema. Whether or not we include such a value constraint, we need to address the fundamental question, which may be phased in ORM terms as: *Is the AccountType instance denoted by the shaded dot identical to the subtype SavingsAccount?* If we answer Yes, then we have a case of an instance being a type in the same model.

In the UML schema, the use of an enumerated type demands a No answer, because UML treats enumeration types as data types whose instances are literals [20, p. 96]. However, as discussed shortly, UML allows us to remodel the situation using a powertype for AccountType, which requires a Yes answer.

As a general point, in specifying a fact type as the application of a predicate to a sequence of object types, we understand that *the predicate applies to instances of the object types*, not the types themselves. For example, consider the fact type Person was born in Country. The "was born in" predicate is understood to apply to ordered pairs of persons and countries (e.g. Niklaus Wirth was born in Switzerland, Terry Halpin was born in Australia). It does not make sense to say the object type Person was born in the object type Country. A similar comment holds in UML when applying associations to a sequence of classes.

[1] In UML 2.0, enumeration types may be extended (by adding further values) in other packages or profiles. This seems inconsistent with a clean approach to subtyping based on substitutability semantics, where subtypes may strengthen but not weaken constraints on their supertypes.

Now let us return to the question as to whether in Figure 1(a) the AccountType instance denoted by the shaded dot is identical to the subtype SavingsAccount. In order for this identity to be even possible, the semantics of the fact type Account is of AccountType should satisfy at least the following necessary conditions:

1. The "*is of*" predicate *means* "is a member of" or "is an instance of", i.e. ∈ (*set membership*).
2. *Only Account instances may be instances of AccountType instances*.

These conditions (which in combination we'll call *homogeneous set-membership*) arguably follow from the *indiscernibility of identicals* (if $a = b$, then a and b have exactly the same properties). The subtype SavingsAccount includes precisely all the possible savings accounts in the business domain—no other things can be instances of it. If we agree that the instance of AccountType denoted by the shaded dot *is* the subtype SavingsAccount, then only accounts can bear the instance-of relationship to it.

If the modeler wishes to view the model in this way, we should allow this interpretation, as it is not inconsistent. On the other hand, we should not force the modeler to adopt this interpretation, as there are often better ways to model such situations that do not require such a commitment to instances being types in the same model (see later discussion).

If we allow the type = instance interpretation, we must use higher-order logic for formalization, and should apply the semantics of homogeneous set-membership to categorization relationships of this kind. If we reject the type = instance interpretation, we may stay with first-order logic (at least for formalizing categorization relationships of this kind), and may optionally distinguish such categorization relationships as special (which some practitioners feel is an important thing to do), and provide them with relevant formal semantics. From a meta-modeling viewpoint, it is trivial to include one or more meta-fact types to classify fact types to cover this case and others.

To note this distinction, one could adopt a special graphical or textual adornment for such categorization associations. For example, in ORM one might append a colon ":" to the forward reading of any predicate used for this purpose (based on the common use of colons to sometimes but not always introduce types). Applying this suggestion to the model in Figure 1(a) would replace "is of" by "is of:".

To provide a minimal, common approach to such categorization relationships, whether or not we adopt the type = instance viewpoint, we could use the colon marker to distinguish any such relationship, and give it the semantics of an *asymmetric, intransitive*, and *locally-homogeneous* relationship. A fact type of the form *A R: B* is *locally-homogeneous* if and only if *B* is used as categorization scheme for *A*, but for no other type (so no other type bears a colon relationship to *B*). For the example in Figure 1, this means that only Account instances may be instances of AccountType instances. It is convenient to use the same predicate reading (e.g. "is of:", or even just ":") for all such categorization predicates, unless this makes the reading awkward. The choice of reading is language-dependent.

The properties of asymmetry and intransitivity seem to be the only properties of the set-membership operator (∈), that are relevant here. If we always use the same reading (e.g. "is of:") for the categorization relationship, we may think of it as a predefined predicate constant that applies globally (all occurrences of this predicate have the same semantics). If we reserve the colon only for such homogeneous cases, we must not use it in cases where the classification scheme (e.g. Gender) may be applied to more than one type (e.g. Person, Dog).

Note that *any* role played by a type could be used as a basis for categorizing it. So the main reason for marking such is-of associations as categorization relationships is to enforce the formal properties of such relationships. We may now formalize categorization relationships of this kind (Figure 2), assuming the same predicate reading "is of:" for all (if this is not so, then without loss of generality, begin by replacing each reading by "is of:"). We use "~" for negation ("it is not the case that"), "&" for conjunction ("and"), and "→" for material implication ("implies"). The asymmetric and intransitive properties may be declared independent of the object types, unlike local homogeneity.

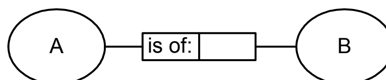

Figure 2 The categorization relationship is *asymmetric, intransitive*, and *locally-homogeneous*.

For the *first-order logic interpretation*, all types are first-order, so instances of B are individuals, not types. We use lower-case letters (possibly subscripted) to range over individuals. For our *higher-order logic interpretation*, we use capital letters (in italics) to denote type variables of any order. The order (1, 2,

3, …) of any type is implicit, since it can be derived by inspecting the full schema[2]. Ignoring the case of crossing meta-levels, assign the order of a type to be 1, plus the number of relationships in a contiguous chain of zero or more categorization relationships that end at the type.

Using ∃! for Stephen Kleene's "there exists exactly one" quantifier, we may now formalize the constraints on the categorization relationship in Figure 3, which has a mandatory and uniqueness constraint. The first-order formalization treats AccountType as a type of individuals. The higher-order formalization treats AccountType as a type of first-order types.

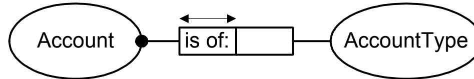

Figure 3 A categorization relationship with two additional constraints.

First-order formalization:

$\forall xy$ [x is of: y $\rightarrow$ ~ (y is of: x)] -- asymmetric

$\forall xyz$ [x is of: y & y is of: z $\rightarrow$ ~ (x is of: z)] -- intransitive

$\forall xy$ [AccountType y & x is of: y $\rightarrow$ Account x] -- local homogeneity (only accounts are of account types)

$\forall x$ [Account $x \rightarrow \exists! y$ (AccountType y & x is of: y)] -- mandatory and unique constraints

Higher-order formalization:

$\forall xY$ [x is of: Y $\rightarrow$ ~ (Y is of: x)] -- asymmetric

$\forall xYZ$ [x is of: Y & Y is of: Z $\rightarrow$ ~ (x is of: Z)] -- intransitive

$\forall xY$ [AccountType Y & x is of: $Y \rightarrow$ Account x] -- local homogeneity (only accounts are of account types)

$\forall x$ [Account $x \rightarrow \exists! Y$ (AccountType Y & x is of: Y)] -- mandatory and unique constraints

Alternatively, the higher-order formalization may replace any expression of the form α is of: β, where β is a type variable, by βα, since it regards α is of: β in such cases to entail that α instantiates the β predicate. This leads to the higher-order formalization: $\forall xY$ ($Yx \rightarrow$ ~ xY); . $\forall xYZ$ (Yx & $ZY \rightarrow$ ~ Zx); $\forall xY$ (AccountType Y & $Yx \rightarrow$ Account x); $\forall x$ [Account $x \rightarrow \exists! Y$ (AccountType Y & Yx)].

UML Powertypes

One motivation for distinguishing such categorization relationships is to facilitate transformation to or from UML models that include so-called *powertypes*, which UML includes specifically to model such categorization schemes. Figure 4 shows a simplified version of an example often used to illustrate the need for powertypes, using the old UML 1.4 notation. This way of classifying trees is botanically wrong, but that's irrelevant to the issue. Let's assume that trees can be classified into species such as Oak, Elm, and Willow in this simple way. Here TreeSpecies is a class analogous to the object type AccountType in Figure 1.

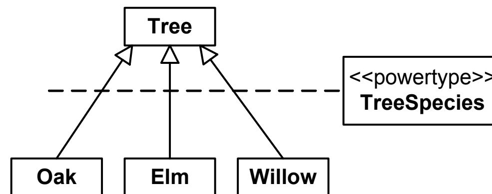

Figure 4 Powertype example in UML 1.4 notation (now retired).

[2] If it is desired to explicitly show the order of a type, a pre-superscript may be used (e.g. ^{2}B indicates B is a second order type, i.e. its instances are types whose instances are individuals). Post-superscripts are typically used to denote arity, and post-subscripts are often used to distinguish variables of the same type.

To indicate that the subclasses Oak, Elm and Willow are each instances of the class TreeSpecies, a dashed line connects the inheritance links to TreeSpecies, which is marked with the stereotype name "powertype". If the name "powertype" derives from the notion of power set (the power set of a set *A* is the set of all subsets of *A*), the term is misleading, as the powertype TreeSpecies excludes many instances in the power set of the set of trees (e.g. the null set, the set of all trees, and many other tree sets). For this reason, the term "*higher-order type*" seems more appropriate than "powertype".

At any rate, this diagram does not explicitly include an association such as Tree is a member of TreeSpecies (analagous to Account is of AccountType). We may treat this association as implicit here, but in practice an explicit version of this association would typically be needed, since we would normally want to know the species of any given tree, and with hundreds of tree species it would be diagrammatically extravagant to introduce subtypes for all of them.

In UML 2.0, powertypes are defined within the superstructure [20, sec. 7.17]. The old stereotype notation has been retired [20, p. 597]. Instead, a colon ":" prepends the powertype name (e.g. ": TreeSpecies") to annotate the collection of displayed subtype-supertype connections that belong to the set of all possible generalization relationships (called a GeneralizationSet) based on the categorization scheme provided by the association that relates the supertype to the powertype. If the subtypes are connected to the supertype using different arrowheads, the powertype annotation is placed next to a dashed line that crosses the relevant subtype connections (see Figure 5a). If a common arrowhead is used, the annotation is placed next to that (see Figure 5b).

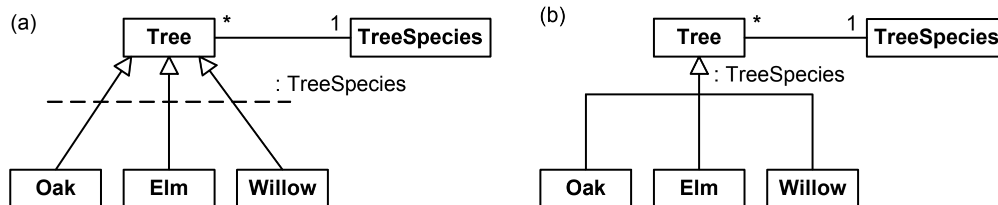

Figure 5 Powertype example in UML 2.0 notation.

Without the old stereotype notation, there seems to be no way to directly indicate that a class is a powertype. If this is the case, we can know that a class is a powertype only if its name used in a generalization set constraint. We may not assume that any binary association from an object type to a type marked in some way as a "powertype" is of this nature (moreover, UML 2.0 retired this stereotype). For example, in addition to Tree is a member of TreeSpecies we might have fact types like Person named TreeSpecies.

Figure 6 shows an example from the UML 2.0 specification [20, p. 127] that provides a different way to model a variation of our earlier account example. Here AccountType is modeled as a powertype rather than an enumeration, and the fact type Account is of AccountType is modeled as an association rather than as an attribute[3]. It is possible to include this association in the model *without explicitly introducing any subtypes* (SavingsAccount etc.). In that case, there is *no way to formally capture the categorization semantics of the association, or to declare that AccountType is a powertype*.

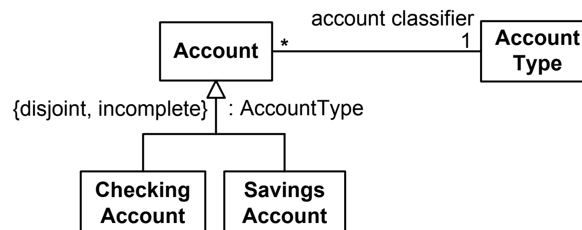

Figure 6 Powertype example from the UML 2.0 Superstructure specification (p. 127).

---

[3] This example differs slightly in allowing more than two account types. If an enumeration constraint holds for AccountType (e.g. {SavingsAccount, CheckingAccount, LoanAccount}), it is unclear how to add this constraint to the powertype, except by resorting to a textual specification of the constraint (e.g. by using OCL).

While the role name “account classifier” *informally* suggests these additional semantics, this has no formal force (other examples in the UML specification use different role names such as “vehicle category” etc.). So *we need to distinguish the categorization association itself*, for example by appending a colon to the relationship reading (e.g. “is of:”) and/or the relevant association role names.

Even if generalization relationships are annotated with the relevant powertype name, this does not always formally guarantee the required semantics. For example, suppose we classify employees as part-time or full-time, and also record their preferred employment status (part-time or full-time) if any. We might model this in UML as shown in Figure 7, with two associations between Employee and EmployeeType. There is no formal way of knowing which of these two associations is used as the basis for membership in the subtypes. To solve this problem, we could require role names in the annotation, or adopt the ORM practice of supplying formal subtype definitions[4].

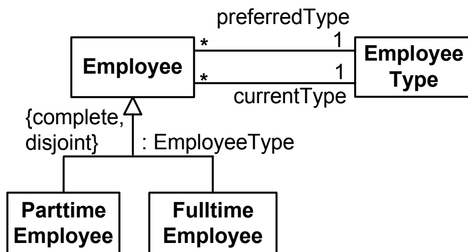

Figure 7 Powertypes do not guarantee an unambiguous classification scheme.

UML does allow generalization annotations to include names for the generalization set (presumably prepended to :powertype names when powertypes are involved, although the UML specification does not clarify this possibility). However such names are treated as informal comments. There is no formal requirement to link these back to properties (attributes or far-roles) of the supertype, as is done in those database modeling techniques that use discriminators to indicate a basis for subtyping.

3 What kind of higher-order logic is appropriate?

The current ORM formalization uses only first-order logic, basic arithmetic, and bag comprehension. Once we adopt higher-order logic, we need to allow any order (not just second-order), because in principle one might always introduce new types to categorize existing types, whether or not those types are first-order. If we adopt standard semantics for higher-order logic, where quantifiers may range over any imaginable predicates, we lose some useful properties of first-order logic. For example, completeness, compactness, and the Skolem-Löwenheim theorems don’t hold in standard second-order logic.

Moreover, care is needed to avoid some well known paradoxes. Russell’s paradox considers the set of all sets that are not members of themselves: is this set a member of itself? If Yes, then No, and if No then Yes, leading to a contradiction. Grelling’s paradox deals with self-predicable or autological properties (properties that apply to themselves). For example, we might argue that the property of being nonhuman is itself nonhuman (and hence is a self-predicable property) whereas the property of being human is not itself human (and hence is a non-self-predicable property). If predicates may instantiate themselves, then using N and H for the properties of being nonhuman and human we might formalize this as NN and $\sim HH$ respectively. But then what about the property of being non-self-predicable? Is this self-predicable or not? If it is, then it is not, and if it is not, then it is. Either way, we have a contradiction.

To avoid such paradoxes, Bertrand Russell developed a type theory in which types are ordered in a hierarchy, and it is meaningful to say that a type is an instance of another type only if the second type is on the next level of the hierarchy. Similarly, predicates of higher-order apply only to predicates or objects of lower orders. In particular, no predicate may apply to any predicate of the same order. Hence no predicate may apply to itself. Essentially the paradoxes are avoided by forbidding predicates to apply to themselves, by adopting a hierarchy of levels in which types can have instances at lower levels only.

[4] The types Employee, ParttimeEmployee, and FulltimeEmployee are time-*deictic*, because their sense is determined in part by the time that their terms are uttered/inscribed [23, pp. 9-10; 17, esp. pp. 304, 312-13]. Deixis has a significant impact on information modeling, but space precludes a proper discussion here.

While this seems a reasonable approach to adopting higher-order logic, there are other versions of higher-order logic that do not take this approach. For example, the logic underlying the Knowledge Interchange Format (KIF) allows predicates to instantiate themselves, so expressions such as (R R) are allowed [16]. An extension of this logic has been proposed as an ISO "Common Logic" standard to facilitate interchange of logical formulae between different knowledge representation tools [21]. Unfortunately, this Common Logic proposal includes a number of "bizarre" properties that make it unintuitive for direct use. Nevertheless, the common logic is very relaxed, and has mappings to several logics, so it seems quite feasible to have mappings between it and a more intuitive logic.

To make the implementation of higher-order logic more tractable, it seems best to adopt a non-standard semantics, similar to Henkin semantics [18, pp. 378-80], to limit the range of predicates/functions over which we may quantify, in order to retain useful properties of first-order logic (e.g. completeness). With standard semantics, a monadic first-order predicate may range over the power set of the domain of individuals (objects: lexical or non-lexical). To deal with categorization-types (e.g. AccountType) where we wish to assert that instances are types, it seems that the only extension we need beyond first-order logic is to allow quantification over object types that are instances of a declared categorization-type (whether or not these instances have been explicitly declared as a subtype).

As a separate decision, if we wish to allow crossing metalevels in the same model (see later), we should allow quantification over object types (primitive or derived), of any order, that are explicitly declared in the schema. If we do allow this, there seems to be no compelling case to allow quantification over polyadic predicates, so this relaxation may be regarded as a restricted case of Henkin semantics.

Given that a move to higher-order logic adds work to the formalization and implementation tasks[5], is explicit support for quantification over predicates worth the extra effort? With respect to the categorization relationship, the only motivations seem to be in noting what kind of relationship it is, in providing direct support for those who view the relationship as an instance-type relationship, and in providing a convenient slot for mapping to/from powertypes in UML. With respect to categorization, these motivations seem non-compelling, given that a first-order interpretation seems reasonable, and formal subtype definitions (as in ORM) provide the connections between subtypes and the predicates and object types used to define them. The next section provides some justification for a first-order interpretation.

4 A first-order logic approach to categorization

Consider the ORM schema in Figure 8, which is a classic case where higher-order logic proponents would demand that CarModel is a second-order type, whose instances are subtypes of Car (e.g. FordFutura2004). When I think of an instance of Person and Car (concrete concepts), I'm thinking of an actual person or car. When I think of an instance of CarModel (an abstract concept), I think of an abstraction that is essentially a car design—a car *structure or specification* that might be denoted by a schematic diagram, for example.

For any given business domain, I define a *type* as a *set of possible instances, where for any given state of the business domain, exactly one subset of the type is the population of the type in that state*. At any given time, the *population* of a type is the set of instances of that type that exist in the business domain at that time. This definition is similar to Fitting's notion of an intensional type as a function from possible worlds to extensions [8, p. 84], except that here a possible world corresponds to a state of business domain, sometimes called a "time-slice" or "instant state" in temporal logic [10, p. 143; 9, p. 121]. The temporal aspect is needed to distinguish between types that have the same *extension over time* but may differ in *extension at some time* (e.g. HumanPerson, and HumanBaby). Note the two senses of "extension". A type's extension at some time is its population at that time. A type's extension over time is effectively the atemporal union of all its state-extensions, past, present, or future, which is fixed. "Type" conveys both state-dependent (variable extension) and state-independent (fixed extension) notions, not just set semantics.

By the Principle of Extensionality, sets are defined by their extension, i.e. given any sets A and B, we say that $A = B$ iff $\forall x(x \in A \equiv x \in B)$. So unless we resort to non-well-founded set theories, we must regard sets to be fixed—they cannot change over time. It seems clear that for any given business domain, the current population of a type (e.g. Person or Country) may change over time, but the type itself does not.

[5] For a plea in favor of higher-order logic, and for some examples where first-order theorems are much easier to prove in higher-order logic see [1].

Given the above definition of type, I personally don't think of an instance of a car model as a set of possible cars of that model (a subtype). Each CarModel instance is in 1:1 correspondence with such a subtype (implicit or explicit), but it's not identical to a subtype. It seems to me that this distinction can always be made. If so, we can treat instances of such "categorization types' as ordinary individuals, that are not types, so first-order logic is enough.

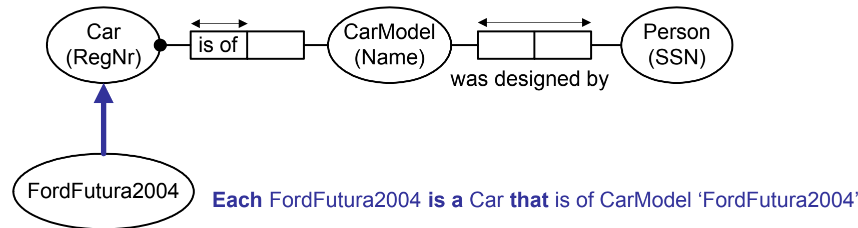

Figure 8 The population of a subtype (or any type) typically varies over time.

Suppose we explicitly introduce a car subtype called "FordFutura2004", as shown in Figure 8. The subtype definition provides the formal connection between the subtype and the car model instance. Using the standard reference mode semantics for ORM, the expression "CarModel 'FordFutura2004'" in the definition abbreviates "CarModel that has CarModelName 'FordFutura2004'". In ORM, a subtype is essentially a derived object type—its population is determined by applying the derivation rule in its definition to the populations of the object types referenced in the definition.

Suppose we add the existential fact **There exists a** CarModel **that** has CarModelName 'FordFutura2004' to the information base before any cars of that model are produced. At this time, the population of the car subtype FordFutura2004 is the null set. Suppose at a later time, 100 cars of that model are produced. At that later time, the population of the car subtype FordFutura2004 includes 100 cars. At any time, the subtype FordFutura2004 includes those 100 cars, as well as all other cars of that model that will ever be produced. So it's perfectly OK to regard an instance of CarModel to be in 1:1 correspondence with a set of (possible) cars. This is true whether or not we think of the car model instance as the set of possible cars of that model, or (as I do) as the fundamental car structure or design to which the car instances conform.

Hence it is reasonable to think of an instance of a "categorization type" such as CarModel or AccountType as an individual (e.g. a structural pattern) that is ontologically distinct from a type (in the sense of a set of possible instances). This seems sufficient to stay with first-order logic for such cases. Although the word "Type" in "AccountType" may suggest that its instances are themselves types, this stems more from an unfortunate naming choice for the type rather than from any fundamental intuition.

The second approach that allows one to avoid higher-order types for categorization is to *avoid uninformative categorization schemes*. The term "AccountType" is uninformative, because it does not provide any basis for categorizing accounts. In principle, any object type such as Account might be categorized in many different ways, leading to different types of bank account. For example, we could define an AccountKind {Local, National, International}, an AccountCategory {Taxable, Nontaxable}, and so on. These are all categorization schemes, which we may wish to use in the same model, and names such as "AccountType" and "AccountKind" don't inform us at all about the criterion used by a given categorization scheme to place accounts into account categories. One might argue that the value constraint placed on the categorization scheme provides this criterion, but this requires the modeler to induce the criterion based on his/her informal understanding of what the names for those values mean, an understanding that is not formally accessible to an automated system. More importantly, we may wish to introduce a categorization scheme without committing to a fixed set of instances.

As a pragmatic issue then, it seems reasonable to encourage the modeler to *choose informative names that reveal the basis for classification schemes*. If we adopt this approach, the type = instance issue typically disappears for the categorization case. In Figure 9 for example, the subtype Savings account is defined based on its primary function. The AccountFunction instance named 'Savings' and denoted by the shaded dot is clearly not identical to the subtype SavingsAccount (a function is not the same thing as a bank account). One may introduce other informative categorization schemes such as Account may be used in Region, etc. Similarly, our Car is of CarModel relationship might be renamed "Car conforms to CarModel".

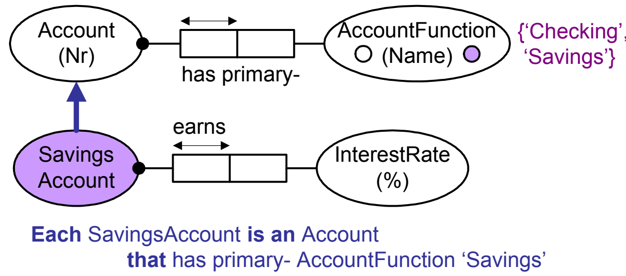

Figure 9 An informative categorization scheme explains the basis for categorization.

Any binary fact type used for an enumerated categorization scheme may be replaced by one or more unary fact types. For example, instead of Account is of AccountType {Checking, Savings}, we may use the fact types Account is used primary for savings and Account is used primarily for checking (the mandatory and uniqueness constraints are then captured by an xor constraint). Instead of Account is of AccountCategory {Taxable, Nontaxable}, we may use the fact type Account is taxable, applying the closed world assumption to determine nontaxable accounts. This is yet another way to avoid higher-order types for enumerated categorization schemes.

In many cases, *an object type may be used to categorize more than one kind of object*. For example, Figure 10 includes two categorization fact types Person is of Gender and Animal is of Gender, where Gender has two possible instances identified by the gender codes 'M' (for male gender) and 'F' for female gender). Here the semantics of the categorization fact types does not involve homogeneous set membership. Clearly, instances of Gender (MaleGender, FemaleGender) may not be identified with any of the four subtypes shown. This kind of categorization scheme is quite common, and clearly excludes any type = instance identities.

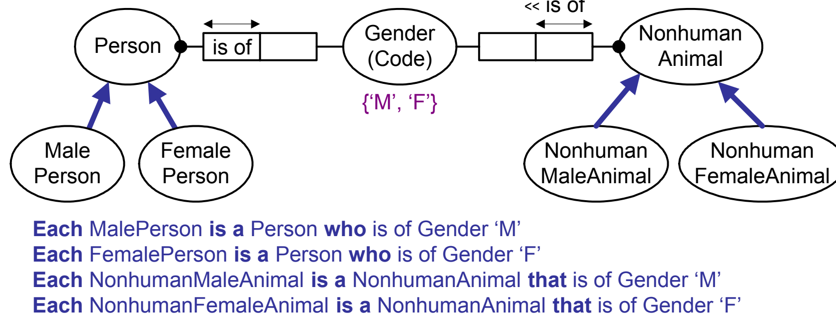

Figure 10 A typical case where instances of the categorization scheme are not identified with subtypes.

The use of Gender in the above model to define the subtypes seems better than using two powertypes PersonType {MalePerson, FemalePerson} and NonhumanAnimalType {NonhumanMaleAnimal, NonhumanFemaleAnimal}. Note that while UML allows this case to be modeled using Gender as an attribute or non-powertype, any use of a generalization set name (e.g. 'gender') is merely an informal comment, with no formal connection to the model element used as the basis of the categorization.

5 Set-structures, and metalevel crossing

Consider a categorization scheme whose instances intuitively correspond to set-like containers. Figure 11 includes a model that, apart from some constraints, is structurally similar to the account example in Figure 1. In this business domain, only A-team members may earn privileges, so a subtype is created to record facts of this nature. The shaded dot denotes team A and the white dot denotes team B. The shaded subtype A-TeamMember is the type whose population at any time is the set of A-team members at that time.

It seems natural to think of Team A at any point in time as more than just a set of people. The concept of a team brings in other semantics (a social unit whose members work together for a common purpose). While the thing denoted by the shaded dot appears to have this additional informal semantics, the subtype A-TeamMember does not—at any point in time its denotation seems to be no more than a set of people who just happen to be members of team A. If these intuitions are correct, and team membership is considered to be a categorization scheme, here is another example where it is reasonable not to identify the subtype with the instance of the categorization scheme.

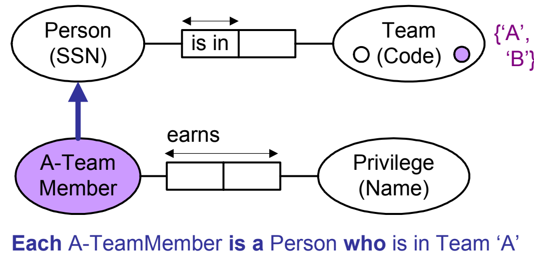

Figure 11 Is the Team instance denoted by the shaded dot identical to the subtype A-TeamMember?

Now consider Table 1, which is a slightly simplified version of an example by Fitting [7], used to motivate a formalization of databases that uses higher-order modal logic. The table has two aspects that are unusual. First, it is in non-first normal form, allowing unnamed sets as entries (e.g. ColorChoices). This is permitted in some object-relational databases. Secondly, its final attribute (column) allows as entries unnamed sets whose instances appear to be attributes themselves, thus crossing levels/metalevels.

Table 1 Table with entries that may be sets of individuals or attributes.

Car	*CarModel*	*ColorChoices*	*AirConditioning*	*CustomerChosen*
1	Ford Escape 2003	{red, green, black}	Yes	{ColorChoices}
2	Ford Escape 2003	{red, green, black}	No	{}
3	Mazda MPV 2004	{green, sand-mica}	Yes	{ColorChoices, AirConditioning}

Fitting's formalization of this situation is higher-order, as he treats the structure directly as it stands. The price paid for this directness is deep complexity, and an implementation nightmare. These disadvantages can be avoided by transformation into a first-order model that is cleaner and pragmatically easier to implement. In practice, it would be realistic to record the color chosen for a car. With this additional fact type, and omitting for now the air-conditioning aspects of the model, the situation may be modeled by the ORM schema shown in Figure 12. Here the color choice sets are handled in the usual normalized way, with a many:many association. The subset constraint (circled "⊆") ensures that colors chosen for a car belong to those available for its car model. The fact whether a customer chose the color for a car is catered for by instantiating the unary predicate applied to the objectified CarColor association. The airconditioning aspects can be catered for in a similar way. For a simpler example of level-crossing, see [6, p. 20].

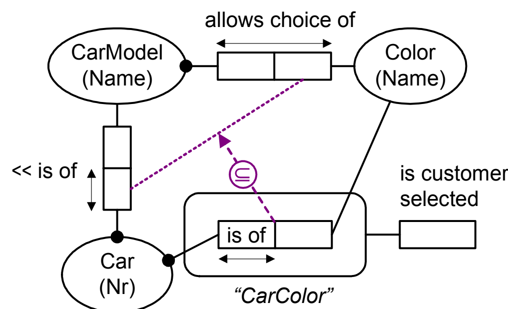

Figure 12 A first-order solution for part of the un-normalized table.

In cases where there are many attributes about which information is to be recorded, and the attributes are not all known in advance, this may be modeled by introducing Attribute as a first-order type, along with fact types that record its name and value. By thus demoting meta-data to data, we remain at first-order. This technique is well-known and quite effective in practice.

A final argument for using higher-order types is to allow expression of business rules that appear to cross levels/metalevels (or ignore level distinctions) in the same model (e.g. the Finance department is responsible for defining the possible values of AccountType). If somebody really wants to formalize such cases directly, then higher-order types are clearly needed. However, as a pragmatic alternative, it is usually possible to handle such rules in a first-order way, either by separating the meta-aspects into a separate first-order model where the former meta-types are now ground types, or by demoting meta-data as data, or simply ignoring the cross-level identities.

As a simple example, we might build into the core package of the metaschema such metafact types as FactRole is played by ObjectType, BusinessRule has IllocutionaryForce etc, while in the management package of the metaschema we include metafact types dealing with aspects of security and authorization etc., e.g. UserGroup has AccessRight to FactType. Populating the latter metafact types in the metamodel may then allow us to add rules of the desired kind without crossing metalevels.

For example, consider the simple business model in Figure 13. Here there are two elementary fact types (F2 and F4), and three existential fact types (F1, F3, and F5) depicted in abbreviated form using parenthesized reference modes. In this case, the current values of AccountType are stored in a reference table instead of being rigidly declared in a value type constraint. One of our business rules is that only the marketing department may choose what the account type instances may be.

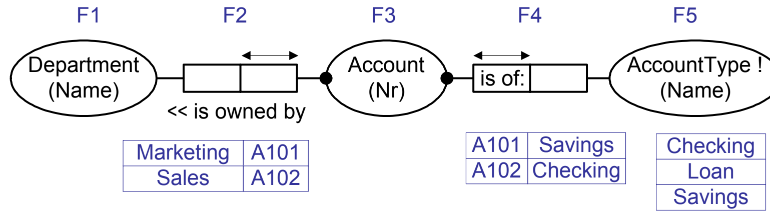

Figure 13 A populated business model with 5 fact types (3 existential and 2 elementary).

With this approach, the one column fact table for AccountType may be treated just like any other fact table in the model, including the way in which we determine who has what kind of access to what fact type. A simplified metafragment for dealing with security is shown in Figure 14. The business rule mentioned earlier is now handled by populating this metafact type as shown.

The metamodel in Figure 14 exists at a level above the business model in Figure 13. Each of these models can be formalized separately, using either first-order logic or the higher-order logic extension for categorization discussed earlier. What is missing from this picture is the ability to identify the marketing department mentioned in the business model with the marketing department that appears as a user group in the metamodel. There does not appear to be any compelling business case to require formal support of this identity, as businesses seem to run perfectly well without such support (e.g. in SQL systems, the application tables and metatables are typically accessed separately).

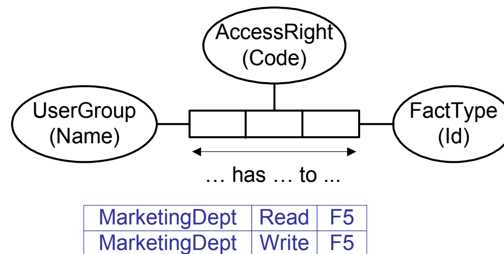

Figure 14 A model fragment from the management package of the metamodel.

6 Conclusion

This paper addressed issues relating to the necessity or otherwise of including higher-order types to deal adequately with three aspects of information modeling: categorization schemes, un-normalized structures, and crossing levels/metalevels within the same model. It argued that higher-order types may be used, so long as the underlying higher-order logic retains certain important first-order properties (e.g. by adopting Henkin semantics). It also suggested a number of ways in which higher-order types may be avoided, by treating types as intensional objects whose instances may sometimes be in 1:1 correspondence (but not identical) to subtypes, by requiring subtype definitions to be informative, by remodeling (including demotion of metadata to data), and by treating types as individuals in separate models.

Various formal properties were established for those categorization types that on first glance appear to require a higher-order solution, and a number of deficiencies were identified in the current treatment of powertypes within UML.

Related future research will investigate the detailed impact of deixis in information modeling, and how issues of semantic equivalence and co-extension are affected by one's stance with regard to the absolute or relative nature of types. For example, when we use terms for object types (e.g. Person, Gender, Country) in modeling, are we thinking only about a given business domain, or a more general concept of the type?

Acknowledgement: The presentation of some ideas in this paper has benefited from discussion with Andy Carver of Northface University, and Don Baisley of Unisys Corporation.

References

1. Andrews, P. B. 2002, *An Introduction to Mathematical Logic and Type Theory: To Truth through Proof*, Kluwer Academic Publishers, Dordrecht.
2. Bakema, G., Zwart, J. & van der Lek, H. 1994, 'Fully Communication Oriented NIAM', *NIAM-ISDM 1994 Conf. Working Papers*, eds G. M. Nijssen & J. Sharp, Albuquerque, NM, pp. L1-35.
3. Chen, P. P. 1976, 'The entity-relationship model—towards a unified view of data'. *ACM Transactions on Database Systems*, 1(1), pp. 9–36.
4. Codd, E. F. 1970, 'A Relational Model of Data for Large Shared Data Banks', *CACM*, vol. 13, no. 6, pp. 377-87.
5. De Troyer, O. & Meersman, R. 1995, 'A Logic Framework for a Semantics of Object Oriented Data Modeling', *OOER'95, Proc. 14th International ER Conference*, Gold Coast, Australia, Springer LNCS 1021, pp. 238-249.
6. Fitting, M. 2000, 'Modality and Databases', *Automated Reasoning with Analytic Tableaux and Related Methods*, Springer Lecture Notes in Artificial Intelligence 1847, Roy Dyckhoff (ed), pp 19--39, 2000. [© Springer-Verlag, URL: http://www.springer.de/comp/lncs/index.html]
7. Fitting, M. 2000, 'Databases and Higher Types', *Computational Logic --- CL2000*, Springer Lecture Notes in Artificial Intelligence 1861, John Lloyd et. al. (ed), pp 41--52, 2000. [© Springer-Verlag, URL: http://www.springer.de/comp/lncs/index.html]
8. Fitting, M. 2002, *Types, Tableaus, and Gödel's God*, Kluwer Academic Publishers, Dordrecht.
9. Girle, R. 2000, *Modal Logics and Philosophy*, McGill-Queen's University Press, Montreal & Kingston.
10. Girle, R. 2003, *Possible Worlds*, McGill-Queen's University Press, Montreal & Kingston.
11. Halpin, T. A. 1989, 'A Logical Analysis of Information Systems: static aspects of the data-oriented perspective', doctoral dissertation, University of Queensland.
12. Halpin, T. A. 1998, 'ORM/NIAM Object-Role Modeling', *Handbook on Information Systems Architectures*, eds P. Bernus, K. Mertins & G. Schmidt, Springer-Verlag, Berlin, pp. 81-101.
13. Halpin, T. A. 2001, *Information Modeling and Relational Databases*, Morgan Kaufmann, San Francisco.
14. Halpin, T. A. 2002, 'Metaschemas for ER, ORM and UML: A Comparison', *Journal of Database Management*, Idea Group Publishing, Hershey PA, pp. 4-13.
15. ter Hofstede, A. H. M., Proper, H. A. & Weide, th. P. van der 1993, 'Formal definition of a conceptual language for the description and manipulation of information models', *Information Systems*, vol. 18, no. 7, pp. 489-523.
16. Hayes, P. & Menzel, C., 'A Semantics for Knowledge Interchange Format', *Proceedings of 2001 Workshop on the IEEE Standard Upper Ontology*, August 2001. The paper itself is available online at: http://reliant.teknowledge.com/IJCAI01/HayesMenzel-SKIF-IJCAI2001.pdf.
17. Lyons, J. 1995, *Linguistic Semantics: An Introduction*, Cambridge University Press: Cambridge, UK.
18. Mendelson, E. 1997, Introduction to Mathematical Logic, Chapman & Hall/CRC: Boca Raton.
19. Object Management Group 2003, *UML 2.0 Infrastructure Specification*. URL: www.omg.org/uml.
20. Object Management Group 2003, *UML 2.0 Superstructure Specification*. URL: www.omg.org/uml.
21. Menzel, C. 'The Common Logic Standard', Online at: http://cl.tamu.edu/CL-ISO.pdf.
22. Rumbaugh J., Jacobson, I. & Booch, G. 1999, *The Unified Language Reference Manual*, Addison-Wesley, Reading, MA.
23. Thomas, J. 1995, *Meaning in Interaction: An Introduction to Pragmatics*, Longman: London.