in UML class diagrams. CT is a branch of mathematics which studies abstract processes [19]. Based on a close connection with formal logic [20], we can use CT to build information models called *ologs* which look very similar to UML class diagrams [21].

Unlike UML diagrams these models are mathematically formal and precise. In later sections we will study the relationships between UML and CT constructions, but for now we merely introduce some historical background and some standard terminology and definitions.

3.1 History

In the late 19th and early 20th century, the character of mathematics shifted radically, moving its focus from classical entities like numbers and shapes to more abstract structures like graphs and vector spaces. As mathematicians developed these abstract domains, they began to see deep connections between different areas. CT was developed as a new language for talking about these sorts of connections.

More specifically, CT was initially developed to understand relationships between algebra and geometry. Though these are very different sorts of mathematical structures, Samuel Eilenberg and Saunders Mac Lane recognized a common framework for representing both sides of the relationship: both support a notion of "structure-preserving" functions [22]. For geometries, these are continuous mappings, whereas in algebra they are usually called homomorphisms (or also linear maps, in the case of vector algebra). The earliest applications of CT were in algebraic geometry and algebraic topology, two areas which bridge algebra and geometry, and quickly became an indispensable tool in those fields [23, 24].

Later, in the 1970's, William Lawvere instigated a new area of study in CT, demonstrating a deep connection between CT and formal logic. He showed that any logical theory can be regarded as a category, and that various logical constructions can be interpreted in terms of categorical structures [25, 26]. It is primarily this connection which we will rely on here.

Following Lawvere's work, CT began to leak out of pure mathematics and into (more) applied fields like computer science and theoretical physics. Today, functional programming is regarded as a best practice in the development of large and complex software systems, and many of the key ideas in functional programming are derived from CT [27]. Its language and methods are part and parcel of some areas of contemporary computer science, most notably the theory of programming languages [28].

CT has also revealed new ways of representing and studying physical phenomena, especially those connected with quantum mechanics. String diagrams, a graphical language developed by Roger Penrose to simplify tensor calculations [29], provide an intuitive yet formal syntax for studying quantum processes. This approach has been especially influential in the area of quantum computing, providing a common base to represent both computational and quantum-mechanical processes [30].

A more recent development is the fledgling emergence of a field of *applied* CT, with a specific focus on concrete, real-world problems. These new approaches, many still under active development, range across a huge swath of application areas including ma-

chine learning [31, 32], dynamical systems [33, 34], database theory [35–37], neuroscience [38] and biology [39], chemistry [40], electrical networks [41], knowledge representation [21, 42] and much more.

As yet, most of this work remains academic mathematics, with a principle interest in proving theorems. Our hope is that this paper will help to make these ideas more accessible, pushing these ideas from theory into practice and broadening the field of applied CT from math and science to include more concrete activities like engineering and design.

3.2 Abstract definitions

A *category* $\mathbb{C}$ is a mathematical structure which encodes processes and the way that they compose. It consists of two types of entities: *objects* $(X, Y, Z, \ldots)$ and *arrows* $(f, g, h, \ldots)$. Each arrow f has an input object (called its *domain*) and an output object (*codomain*) which we indicate by writing $f : X \to Y$. Intuitively, we think of f as a process which takes elements of X as inputs and returns elements of Y as outputs.

Objects and arrows determine a directed graph. A category enhances this structure with a *composition* operation. If $f : X \to Y$ and $g : Y \to Z$, then the output of f matches the input of g; feeding one into the other, we obtain a new process $f.g : X \to Z$. Together with a few more technical assumptions (identity arrows, associativity, see [19]), these rules define the structure of a category.

In understanding the meaning of a category, we frequently rely on a set-theoretic interpretation: objects are sets and arrows are functional (many-one) relations. In this case, composition is defined by the usual formula $(f.g)(x) := g(f(x))$. However the CT approach is quite flexible, also allowing for examples arising in algebra, geometry and probability. Definitions and theorems expressed in CT can then be specialized to any of these areas.

A good example is the notion of an isomorphism. An arrow $f : X \to Y$ is an *iso* if there is another arrow $g : Y \to X$ such that $f.g = \mathrm{id}_X$ and $g.f = \mathrm{id}_Y$. Intuitively, f provides a dictionary allowing us to translate back and forth between X and Y. This formal definition simultaneously generalizes many important mathematical definitions including bijective functions and invertible matrices.

Isomorphism in CT plays a role analogous to equality in classical mathematics; we write $X \cong Y$ to indicate that two objects are isomorphic. By explicitly framing equivalence in terms of invertible mappings rather than simple equations, translation rather than identity, CT manages to avoid some common pitfalls like name-space collisions between related model elements. The remainder of the paper will explore these and other CT methods through their relationship with analogous constructions found in UML class diagrams.

4. From UML classes to objects and arrows

In UML, the *class diagram* is used to describe the key conceptual entities in a context of interest, as well as the important relationships between them. The entities themselves are named, establishing a common terminology, and also described (implicitly) by a collection

of attributes and operations. The terminology and viewpoint derive from UML's origins in modeling for object-oriented programming (OOP), where a class is defined by the attributes and operations that it implements. The principal difference between an attribute and a operation has to do with the way they are implemented on a computer; an attribute stores static (though mutable) information while an operation performs calculations and manipulates data.

In this section, we talk about translating UML classes into the language of objects and arrows. In general terms, this is easy: classes are objects and attributes/operations are arrows. For now we will adopt two tenets of functional programming: immutability of attributes and purity of operations. This simplifies the translation by eliminating issues of state-dependence; we will relax these assumptions later on, in section 4.2.

4.1 Classes, attributes and operations

A *class* in UML is a collection of entities which are conceptually related, often because they share some common properties of state, behavior, etc. The class itself is a collection, and the individuals that make up the collection are called *instances*.[1] Although a class represents a collection of individuals, by convention, class names are capitalized singular nouns. Instances of the class are uncapitalized, and membership in a class is indicated by a typing assertion like `mike:Employee`.

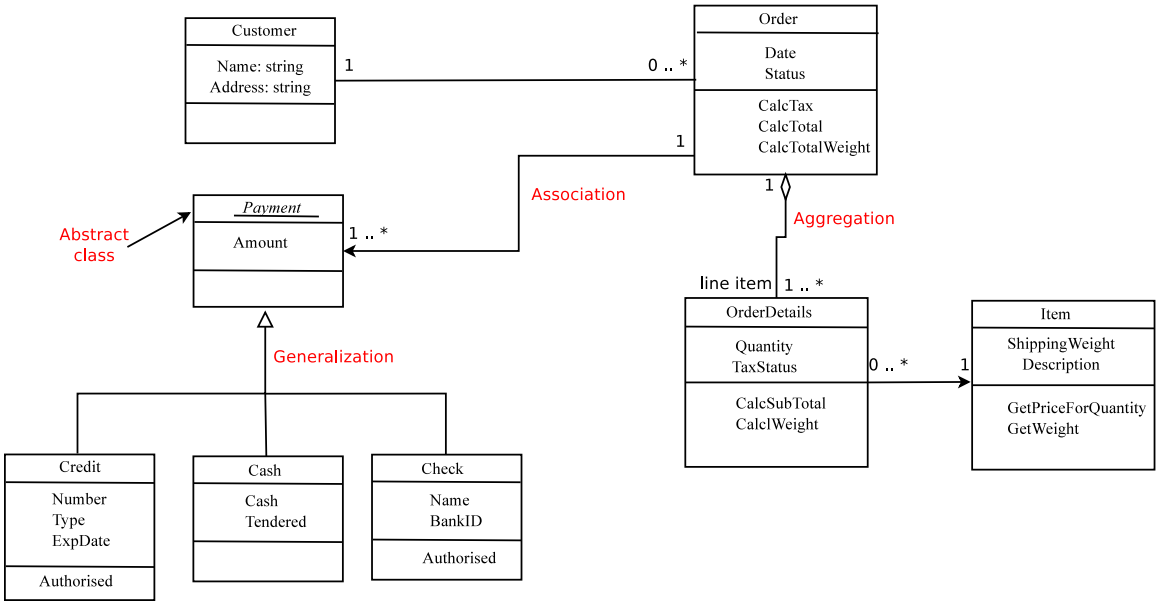


Fig. 1. UML class diagram for analyzing the relation between a Customer and an Order [1]

[1] In OOP (though not in UML) these are usually called "objects", but we avoid this usage because of the conflict with CT terminology.

In CT, objects generalize the role of classes. In particular, when translating from UML to CT every class will determine a CT object (though not vice versa). In practice, CT objects are labeled by names just like UML classes, but these names are purely conventional; they convey an intended interpretation for the model but play no role in its formal specification. Other objects in our CT models are not derived from UML classes, such as data types like `String` and `Integer` as well as more complex data structures defined by operators such as Cartesian products or list constructors.

UML also supports a distinction between abstract and concrete classes; the names of abstract classes are rendered in italics like the *Payment* class in figure 1. This distinction is rooted in the OOP paradigm, where abstract classes have looser implementation requirements than ordinary classes. Our translation to CT will erase this distinction, as it pertains to the specifics of OOP, although we find that in some cases the abstract designation hides a more structural relationship underneath. We will return to this issue in section 5.2.

Each UML class is further elaborated through a collection of attributes and operations located inside the two interior boxes of each class, as shown in figure 2. In OOP, attributes represent stored data while operations describe dynamic computations. Both attribute and operations return values, and may be assigned an optional type (e.g., `Int`, `String`) to restrict these values. Attribute and operation names are usually written in lower case, to distinguish them from class names.

More specifically, an *attribute* describes a characteristic that all members of a class (are assumed to) share. For example, we might specify that every `Employee` has a `name:String` and a `wage:Float`. We write `mike.wage` to refer to the value of the `wage` attribute for the particular class instance `mike:Employee`.

In order to simplify the initial presentation, in this section we will assume that all attributes are *immutable*, meaning that their values are fixed and do not change over time. We will lift this restriction in the next subsection, when we talk about state. Each instance of the class defines a vector of attribute values, and these vectors define a state space for class instances. Immutability means that the location of an instance in state space does not vary over time.

In CT, each attribute defines an arrow whose domain is the class itself and whose codomain is the type of the attribute. This means, first of all, that attribute types like `Int` and `String` must be objects of our CT model. Implicitly, we also include arrows corresponding to type operations like addition and concatenation, though we do not draw them in our diagram unless we need to use them. Now we can then think of an attribute as a functional relationship where the input is the class instance itself and the output is the attribute value; thus the `wage` attribute of the `Employee` class determines an arrow

$$\texttt{wage} : \texttt{Employee} \longrightarrow \texttt{Float}.$$

If an attribute is untyped, we can associate it with type `Any` with minimal assumptions. However, in most cases the type of an attribute is clear from context, and CT encourages us to make this knowledge explicit.

An *operation* represents an action or computation that all instances of a class can perform. Such a computation determines a mapping from inputs to outputs, which we represent as an arrow in a CT model. From the CT perspective, the primary difference between attributes and operations is that the latter may depend on additional input variables. For example, we might want to know how many hours an employee worked on a given day, suggesting a method `hoursWorked(day:Date):Float` which takes the date of interest as a parameter.

Notice that the result of this operation depends on both the employee (i.e., the class instance) and the date, so both should be regarded as inputs to this function. For now we will assume that operations are *pure*, meaning that (for a given instance) the operation will always return the same output when given the same input. Purity of operations is somewhat analogous to immutability of attributes, and we will lift this restriction when we discuss stateful classes.

When we represent a operation as an arrow, its codomain (target) is its declared type, just as for attributes. The difference is that the domain (source) of a operation will involve both the declaring class (here `Employee`) and the class of the explicit input (`Date`). We can combine the two using the Cartesian product, an operation which generalizes from set theory to CT. Thus the `hoursWorked` operation should define an arrow

$$\texttt{hoursWorked} : \texttt{Employee} \times \texttt{Date} \longrightarrow \texttt{Float}.$$

For a operation with multiple arguments, the domain simply includes more factors in the product.

It is also worth mentioning the CT interpretation of so-called "static" operations in OOP. A *static* operation is one which does not depends on the state of the instance that executed it. In other words, the value of a operation depends *only* on its explicit input values. Thus, in the CT translation, this corresponds to an arrow which *omits* the declaring class from its arguments.

For example, we might have an operation $\texttt{dailyPayroll}(\texttt{day} : \texttt{Date}) : \texttt{Float}$ which adds up the pay for all the employees who worked on a given day. Anyone who calculates the payroll should get the same result; since this value is independent of the invoking instance, this is a static operation. In CT, it corresponds to an arrow

$$\texttt{dailyPayroll} : \texttt{Date} \longrightarrow \texttt{Float}.$$

Compare the form of this arrow with that of `hoursWorked`, above, which is not static.

Another collection of attribute/operation modifiers—`public, private, package, protected`—concerns access control. These declare which elements of a system are allowed to see the value of an attribute or invoke a given operation. Just like the distinction between abstract and concrete classes, access control is an issue that derives directly from OOP, and in UML it is modeled as an issue of namespace management. We have already noted that naming conventions are orthogonal to the mathematical structure of our CT models and, in keeping with our goal of abstracting OOP features away from our models, we

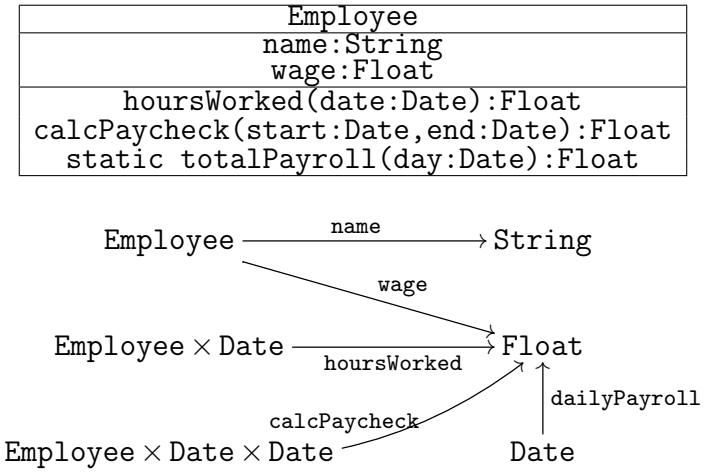


Fig. 2. A (stateless) UML class and the corresponding CT model.

will omit these modifiers in the present discussion.[2]

We now have a rubric for translating any immutable, pure class model into a CT model. An example of such a translation is shown in the Figure 2. We note two specific points.

First, although the CT model seems more complicated than the original UML class, this is only because the CT model makes some details of the original more explicit. The clearest example is our insistence that every attribute and operation should provide a return type, corresponding to the codomain of the associated arrow. Similarly the CT model is explicit about the way that attributes and operations depend on class instances, which appear as factors in the domains of their associated arrows. This dependence is obvious in context–there is no `name` without an `Employee`–but in the UML model it is left implicit. The transparency of the CT model, in turn, helps to formalize the meaning of other constructions like the `static` keyword.

Second, CT also flattens out certain distinctions in UML, leading to a simpler metatheory. Both attribute and method are collapsed into a single concept: the arrow. Some distinctions between these concepts are important, such as the potential for auxiliary inputs, but these can be modeled using existing, general-purpose constructions like Cartesian products. Others distinctions, such as the differences in OOP implementation, are too domain-specific to include at this level of abstraction. The same holds for the distinction between abstract/concrete classes and access modifiers. Generally speaking, CT simplifies the theory of our modeling language by encouraging parsimony in theoretical constructs.

[2]One way to approach this issue in CT is to think of each categorical model as an interface. A `private` access modifier splits this single interface in two. First there is the `Private` interface which contains the entire model, together with a `Public` sub-interface which omits (the arrows corresponding to) any `private` attributes and operations. The "sub-interface" relationship is mediated by a categorical mapping called a functor `Public` $\to$ `Private`. A careful study of access is beyond the scope of this paper.

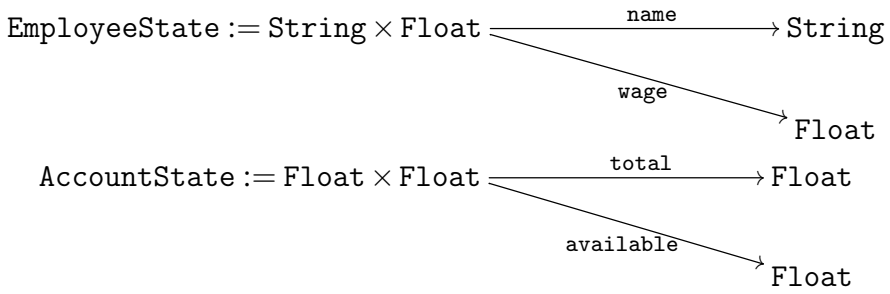
4.2 State

The last section describes a fairly simple translation of UML classes into CT objects and arrows. However, this simple translation depended on some assumptions–immutability and purity–which say that an instance's state and behavior do not change over time. In practice this is unrealistic; in many cases we *expect* attribute values to change over time, and for methods to both depend on and modify these state parameters.

Consider the simple example of a (bank) `Account` class shown in figure 3. The state of an account is determined by two quantities: the current balance in the account and the (possibly smaller) amount which is available for withdrawal, corresponding to the `total` and `available` attributes, respectively.

Whereas an immutable attribute would define an arrow $\texttt{total} : \texttt{Account} \to \texttt{Float}$, a mutable attribute will determine a whole family of such functions, indexed by time: $\texttt{total}_t : \texttt{Account} \to \texttt{Float}$. Equivalently, we can describe this as a single map $\texttt{total} : \texttt{Account} \times \texttt{Time} \to \texttt{Float}$.

At this point it is useful to shift focus from individual attributes to a more general notion of *state*. Any class determines a *state space* which is the Cartesian product of the types of its attributes. We can use projections to recover individual attributes from the larger state space. In our categorical models, every class `Cls` will determine two objects: the class itself `Cls` and its associated state space `ClsState`. So, in the two examples we have seen so far



Now we can track the state of each instance over time as a single function

$$\texttt{state} : \texttt{Account} \times \texttt{Time} \longrightarrow \texttt{AccountState}$$

For example, we might have two bank accounts A and B, initially containing \$500 each (all of which is available). In sequence, B deposits a \$100 check from A, then B withdraws \$50, and finally the deposited check clears. The time-indexed state functions can be tracked using a table whose entries are points in state space (i.e., containing , as below:

`state=(tot.,av.)`	Initial	Deposit	Withdrawal	Clear
A	(\$500, \$500)	(\$500, \$500)	(\$500, \$500)	(\$400, \$400)
B	(\$500, \$500)	(\$600, \$500)	(\$550, \$450)	(\$550, \$550)

Although each instance defines such a time-indexed function, tracing out its internal state over time, we cannot use this map as an element in our model: it corresponds to the ground truth behavior of our instances, and we typically do not have direct access to this information.

To see this, it is useful to note that different purposes entail different relationships with this ground truth mapping. In design we wish to predict and constrain state values; in operation, we monitor and update them; in documentation, we store, organize and analyze them. In particular, our UML model does not contain enough information to derive this behavior over time, especially given its dependence on stochastic processes like bank withdrawals.

Instead of tracking the ground truth, a class description provides a sort of dynamical model for the class, describing (i) a state space for the class, (ii) initial values for new instances and (iii) possible state transitions. These correspond to the attributes, default values and methods of the class, respectively. Given this information *and* a history of interactions (i.e., method invocations), we can then derive the state of the system over time.

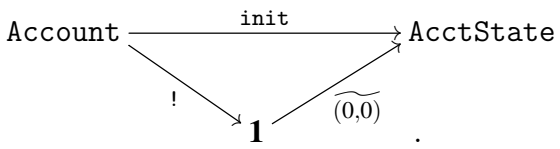
Transitioning from stateless to stateful classes requires three important modifications to our CT translation. The first we have already mentioned: we associate each UML class `Cls` with *two* objects, `Cls` and `ClsState`. An elements of the first is an instance of the class; an element of the second is a potential state that an instance may occupy, and is defined by the attributes declared in the class.

Second, we must define initial conditions for our dynamics. Intuitively, this should be an arrow

$$\texttt{init} : \texttt{Cls} \longrightarrow \texttt{ClsState}.$$

There are at least two common ways to provide initialization. One is to assign a default state like ($0, $0), with the assumption that all new accounts will begin in this state. In CT, we represent a single element of a set $a_0 \in A$ as an arrow $\widetilde{a_0} : \mathbf{1} \longrightarrow A$. Here the object $\mathbf{1} := \{*\}$ is a one-element set, and we can recover the original element of A from the arrow by evaluating $\widetilde{a_0}(*) = a_0$.

The one-element set is called a *terminal object* and has the following property: for any set B, there is exactly one function $B \to \mathbf{1}$ (often denoted by `!`), sending every element $b \in B$ to the unique element $* \in \mathbf{1}$. Whereas the original function $\widetilde{a_0} : \mathbf{1} \to A$ represents a constant element $a_0 \in A$, the composite $B \to \mathbf{1} \xrightarrow{\widetilde{a_0}} A$ can be thought of as a constant *function*, sending every element $b \in B$ to the same place: $a_0 = \widetilde{a_0}(*)$. In the present circumstance, we can use this construction to represent a default initial state as a constant function



Alternatively, we can introduce a method to create new instances, often based on the value of some auxilliary parameters. For example, we might want a method that creates a new account based on a given deposit amount. This would correspond to an arrow

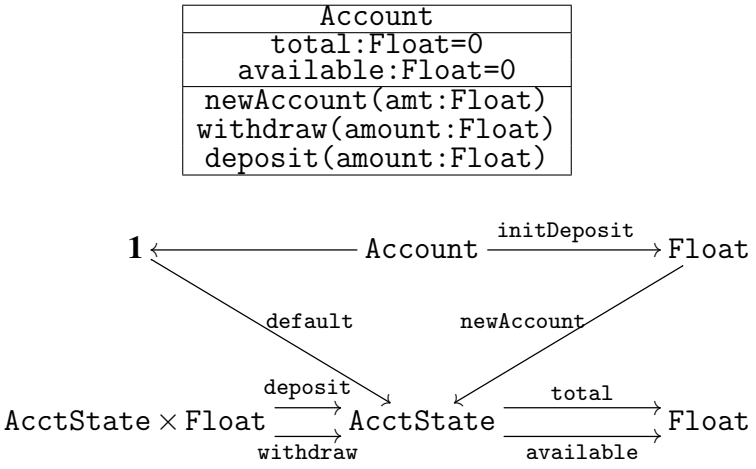
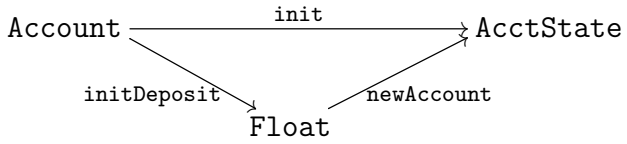


Fig. 3. A (stateful) UML class and the corresponding CT model.

`newAccount` : `Float` → `AccountState`. This produces initial states in much the same way as above, with the additional caveat that each account must be associated with an attribute capturing this initial investment:



Either way (and perhaps both), we end up with an initialization arrow from each class into that class's state space.

The last significant modification to our simple translation in the last section concerns the methods of the class, but there are several differences. First, a method will now depend on an instance's state rather than the instance itself. In terms of CT, the `Cls` factor in the domain of a method is replaced by `ClsState`.

The second and more substantial change is that methods may now modify the state of an instance. This has several consequences. For example, consider the `deposit(amt:Float)` method shown in figure 3. This takes as arguments a deposit amount and (implicitly) the current state of the bank account, and returns a new state. Thus we can think of the `ClsState` object as both an input and an output to this arrow; the amount becomes an additional input and, because no value is returned, the state is the only type of output.

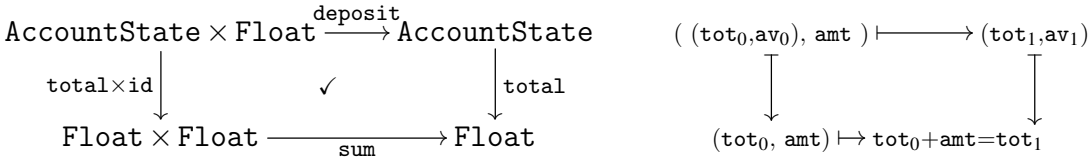
For other methods which both modify state and return a value, we would package these together using Cartesian product, just as for inputs. More formally, an arbitrary method in `Cls` of the form $\texttt{method}(\texttt{input} : A) : B$ should be interpreted as an arrow $\texttt{ClsState} \times A \to \texttt{ClsState} \times B$. With this, we now have everything we need to convert a stateful UML class into a explicit CT model. A simple example is given in figure 3.

In addition to describing the components which make up our class, we might also want to specify some constraints which limit their behavior. If we deposit $50 into an account

with $125, the resulting total should be $175. Though UML does not include such statements directly, there is the OCL extension which provides a logical syntax for specifying constraints [43].

In CT, we include these additional semantics using *commutative diagrams*. Roughly speaking, each directed path through a diagram defines a function, and a commutative diagram constraint specifies that each of these definitions result in the *same* function. For example, the total balance after a deposit should be the sum of the original balance and the deposit amount. Sometimes we distinguish commutative diagram constraints from other, non-commuting diagrams by drawing a ✓ in the diagram.

This constraint is shown as a diagram below. The square on the left is the commutative diagram defining the constraint; the square on the right traces an instance through the diagram to reveal the associated equation.



Similarly, we could demand that the available balance after a deposit is unchanged.

The constraints associated with `withdrawal` are a bit more complicated, due to the if/then nature of the action (i.e., $\texttt{total} \overset{?}{>} \texttt{amt}$). This too can be handled as a commutative diagram, but doing so requires a formal construction called the coproduct which we will introduce in section 5.3.

The explicit declaration of commutative diagram constraints has a nice side-effect: the databases that we produce from these models are automatically normalized. Normalization is a body of techniques which can be applied to reduce redundancy among values in a database, simplifying the maintenance of data integrity. For example, if the same semantic value appears in two different places in a database, changes in one location must be propagated to the other to maintain consistency. Commutative diagrams make these constraints explicit so that they can be built into database systems directly rather than hand-coded in an ad hoc fashion. See [44] for more details on categorical databases.

Now that we have a reasonably complete picture of individual UML classes, we are ready to consider associations, UML relationships which allow us to link different classes to one another.

5. Modeling UML Associations with Arrow Diagrams

While the class diagram provides important information about the structure of individual classes through its attributes and methods, the real purpose of a class diagram is to describe the important relationships *between* the classes that it describes. In UML, these relationships are called *associations*. In this section we will see that associations can often be decomposed into simple diagrams of functional relationships, allowing us to encode these

associations into the language of objects and arrows. The multiplicities for these associations can (usually) be reformulated in terms of special arrows called monics and epics, which generalize set-theoretic injections and surjections. These monic arrows also provide for an interpretation of a separate category of special UML associations called generalizations. We also introduce a new CT construction, called the coproduct, which provides a concise description and additional semantic information associated with many real-world examples of generalization.

5.1 Association

An association represents a specific relationship which may exist between (the instances of) two given classes. In a UML class diagram, associations are indicated by line drawn between the two classes, as shown in figure 4. Customers are associated with the orders which they place, and students are associated with the courses that they study. Often, the ends of an association will include decorations which specify numeric constraints between related members; the example in figure 4 indicates that a customer may place any number of orders (including zero) and that every order is placed by exactly one customer.[3] We may optionally assign names to an association. These names are directional, and may go in either direction or both: the same information is encoded in the relationships "Student a studies course b" and "course b enrolls student a".

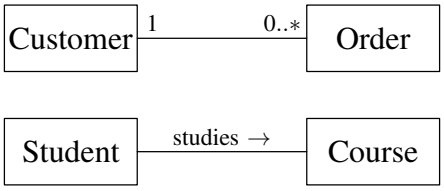


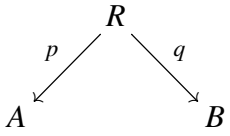
Fig. 4. Two associations between classes in UML

Arrows in CT represent functional (many-one) relationships, whereas associations in UML represent arbitrary (many-many) relationships, so the latter would seem to be more expressive. This is true if we limit ourselves to a single arrow, but pairs of arrows can be used to represent arbitrary relationships (and more). Understanding this encoding is our goal for this section.

Mathematically speaking, a relationship between two sets A and B can be encoded by a subset of the Cartesian product $R \subseteq A \times B$. If a is related to b (under the given relation), then $(a,b) \in R$; otherwise, not. If we restrict the product projections to R, we obtain a pair

[3] In fact, UML attributes may also be assigned multiplicities, and these may be handled using the same methods discussed here for associations.

of arrows as shown below.



In CT, a diagram of this shape is called a *span*, R is its *vertex*, and p and q are its *legs*. This allows us to encode arbitrary relations into objects and arrows. This procedure of adding a new object corresponding to the relation is sometimes called *reification*; in UML we can mimic this approach using *association classes*, which let us attach attributes and operations to elements of an association.

To understand this encoding, it is helpful to first go in the opposite direction: what would an arrow look like in UML? A function $f : A \to B$ represents some sort of relationship between A and B, so we would expect to represent it as an association between the classes A and B. The question then becomes, how can we specify that an association is functional?

Recall that a function $f : A \to B$ is a mapping that associates *every* element $a \in A$ with a *unique* element $b = f(a) \in B$. We have already seen this pattern above, when every order was associated with exactly one customer; generalizing from that specific case, we can always indicate functionality (from A to B) by decorating the B-end of the association with the number 1. Alternatively, this means that we can represent the association between orders and customers as an arrow

$$\texttt{Order} \longrightarrow \texttt{Customer}.$$

Note the reversed direction.

Once we have recognized the above relationship between numerical decorations and functionality, it is natural to ask what the other decorations might indicate. Clearly, if both A and B are decorated with a 1, as in the first row of figure 5, then the relationship is a function in either direction. This is a very strong condition, and is in fact equivalent to the categorical concept of *isomorphism* (iso arrow) which we introduced at the end of section 3.

Diagrammatically, we indicate that an arrow is iso by decorating it with a tilde, as in $i : A \xrightarrow{\sim} B$. Unlike other arrows, there is a symmetry between isos and their inverses, and we often emphasize this symmetry by writing $A \cong B$, rather than specifying a direction for the arrow. As the notation indicates, isomorphism in CT plays a similar role to equality in more traditional mathematics.

When we decorate the A-end of a functional association with the numbers $0..*$, this places no additional restrictions on the relationship: a customer may have any number of orders. By contrast, if we decorate it with the numbers $1..*$, this places a restriction on the association: every $b \in B$ must be the image of at least one $a \in A$. This is the definition of a *surjective* or *onto* function in set theory.

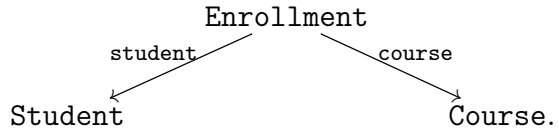
In CT, a similar role is played by the notion of an *epic* arrow (epi), indicated notationally by a double arrowhead $A \twoheadrightarrow B$. The formal definition of such a map is just beyond the scope of this paper, but it is important to note that this concept is derived from the language

of objects and arrows, rather than representing a new construction. This is important semantically speaking because new concepts must be supplied with interpretations, whereas the interpretation of a derived concept comes for free with its definition. Regardless, the intuition quite natural: an epi is an arrow whose image covers the entire codomain.

Alternatively, we could instead decorate the A-end of a functional association with the numbers 0..1; every b is associated with *no more than* one a. This institutes very different restriction on the mapping: distinct a's map to distinct b's. This corresponds to the set-theoretic notion of an *injective* function, which generalizes in CT to a *monic* arrow. Monics are usually indicated by an arrow with a tail, as in $A \rightarrowtail B$ or $A \hookrightarrow B$; monics, like epics, are a derived concept.

Now we would like to use these concepts to model non-functional relationships. As mentioned above, the key is to represent the association as a span, a pair of arrows with a common domain. An element of the common domain is just a pair of related elements; we think of these as elements of the association.

For example, we could model the association from the second line of figure 4 by introducing a new object `Enrollment` together with a pair of arrows



What about its multiplicities? A span is functional just in case one of its legs is an isomorphism; the span itself is an isomorphism if *both* legs are. Since a student may enroll in several courses, and a course enrolls many students, we do not expect either direction of this association to be functional. In other words, neither leg of this span is an iso.

We may also ask whether the legs are monic or epic. To say that the projection from `Enrollment` to `Student` was monic would mean that every student was enrolled in at *most* one course. Similarly, the course projection would be monic if each course could enroll at most one student; both of these are clearly unreasonable restrictions.

On the other hand, it might make sense to require that every course should have at least one student. This corresponds to a model in which the projection to `Course` is epic. Similarly, the student projection is epic if every student must be enrolled in at least one course.

Now we are ready to identify a general procedure for transforming a UML association into objects and arrows. A table containing these translations is given in figure 5.

First, assign a name to the association and create a new object with that name (R, in the figure). Also create two projection arrows from the new object into the classes which participate in the association. Exactly the same procedure can be used to model associations between three or more classes.

Next, associate each multiplicity decoration with an arrow type. Here the decoration "1" corresponds to isomorphism, "0..1" to monic, "1..*" to epic and "0..*" to an ordinary arrow (no constraint). Confusingly, the decoration written on one side of the association

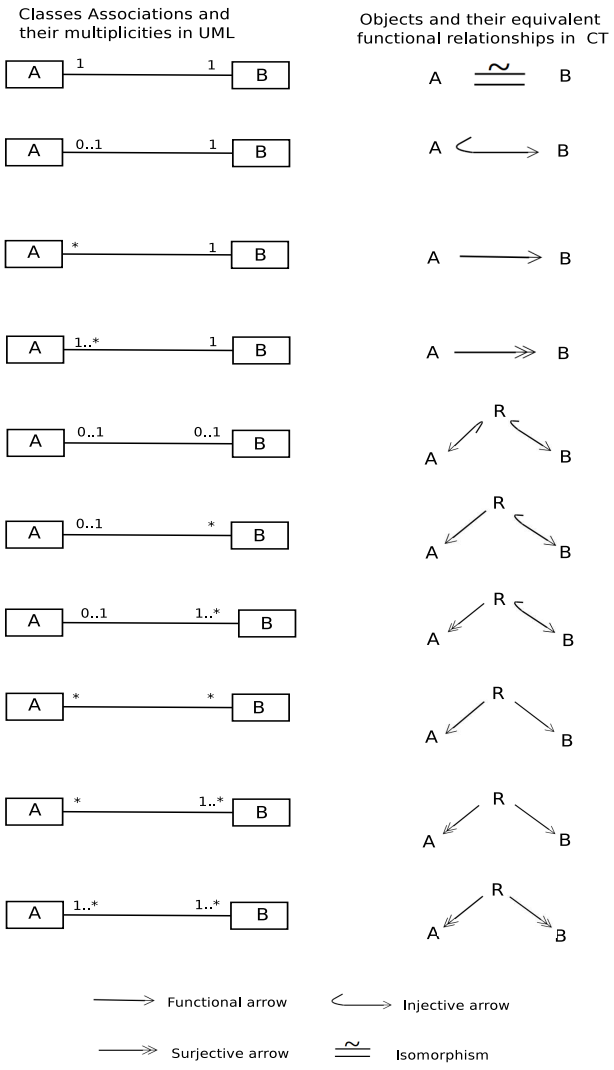
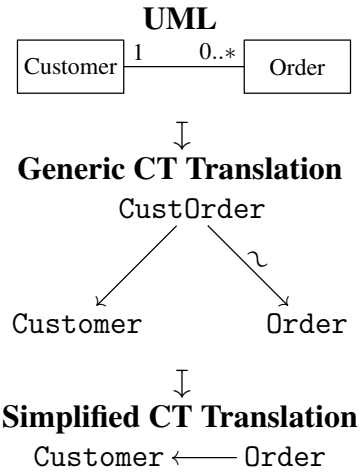


Fig. 5. A translation from UML associations with multiplicity into CT spans.

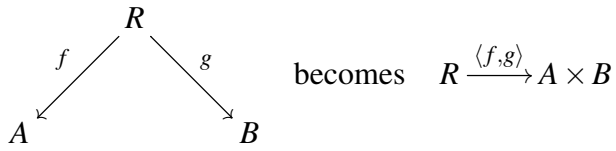
matches with the arrow pointing in the opposite direction, so that a "1..*" decoration on the A-end of an association corresponds to an epic arrow $R \twoheadrightarrow B$. in the span of arrows.[4]

Finally, if one leg of the resulting span is an isomorphism we may remove that leg, leaving just one arrow connecting the two classes directly. For the case for the Customer/Order association, this recovers the same answer that we arrived at above.

[4] More generally, UML allows arbitrary numeric upper and lower bounds on associations. Modeling these requires some additional idea; see Section 6 for details.



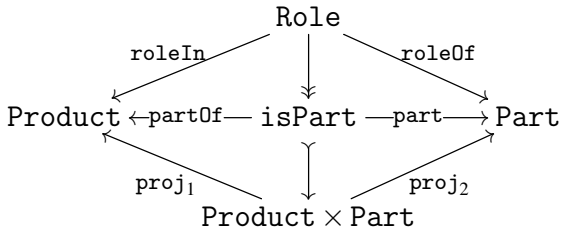
There is one additional point to make regarding the interpretation of UML associations as categorical spans. Using the Cartesian product, we can encode both legs of a span into a single arrow:



In CT, we say that f and g are *jointly monic* if this paired mapping is a monic arrow, in which case we also say that the span is a *relation* between A and B. This is a slight categorical generalization of the usual mathematical definition of a relation as a subset $R \subseteq A \times B$.

An example is helpful to understand the way that spans and relations differ. Consider the relationship between a family of products and the parts from which they are constructed. A relation can tell us whether or not product p contains part q; a span can tell us *how many* q's belong to p.

We might call the first relation `isPart`, while the more general span describes the `Role` of various parts. Clearly the relational model can be derived from the span model–q is a part of p just in case q has at least one role in p–and this relationship is mediated by an epic arrow from the span to the relation, as in the diagram below:



Since UML associations are usually assumed to be relational, we should add this restriction into our categorical translation. In practice it often turns out that the relational assumption is not needed explicitly, even though it may be useful for building intuition.

In that case we can use the less constrained structure of a (not-necessarily-relational) span to model the association. We lose very little by doing this: a relation is a special kind of span, and we can add in the relational assumption later if we find that we need it.

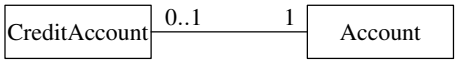
5.2 Generalization and monics arrows

The UML class diagram provides a special construction, called *generalization*, to indicate that one class A is a subclass of another class B. In other words, B is a generalization of A if every instance of A is necessarily an instance of B.

Diagrammatically, generalization is indicated by an arrow with a hollow tip, pointing towards the superclass, as shown in figure 6.

In CT, it is common to model subclass/superclass relationships with monic arrows. This is the same construction that we met in the previous section in modeling class associations, hinting at an unnecessary replication of concepts.

In particular, the discussion of the previous section showed that a special class of associations–those decorated by the multiplicities 1 and 0..1–can be modeled by a single monic arrow:



becomes $\texttt{CreditAccount} \rightarrowtail \texttt{Account}$

In words, we might describe the difference between generalization and monic association as follows:

- With generalization, every `CheckingAccount` *is* a `Account`.
- With a monic association, every `CheckingAccount` *is associated with* a unique `Account`.

Mathematically speaking, this is the difference between a subset $A \subseteq B$ and an injective function $i : A \rightarrowtail B$.

From the perspective of CT, though, this is largely a distinction without a difference. If $A \subseteq B$, we can define an injective function $i(a) = a$. Vice versa, the image of an injection defines a subset $i(A) \subseteq B$ and, by injectivity, $A \cong i(A)$. In either case, we get a pointer from each credit account to a unique bank account, and we can follow this pointer whether or

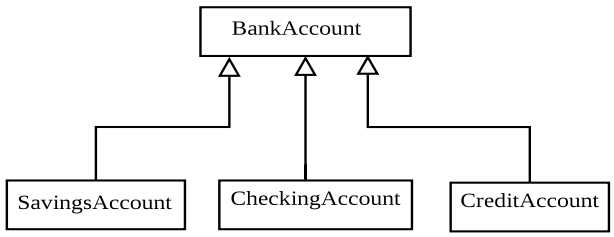


Fig. 6. A generalization relation between two UML classes

not the credit account is "really" a bank account or not. CT encourages us not to focus on what an entity *is*, and rather what it *does* as encoded in its relationships with other entities.

There is an important caveat: while there is only one way to have $A \subseteq B$, there may be different monic arrows $m \neq m' : A \rightarrowtail B$. This arises, for example, in the case of multiple inheritance. Consider a diamond in which both B and C generalize A, and D generalizes both B and C. If generalization is interpreted using subsets this exhibits A as a subset of D. Using monics, we may have two *different* embeddings $A \rightarrowtail B \rightarrowtail D$ and $A \rightarrowtail C \rightarrowtail D$. The first interpretation is more intuitive, but the latter is more flexible, as we can demand coherence between subclasses using commutative diagram constraints.

There is a large and complex literature concerning subtyping; see [45] for an overview. In fact, a CT analysis can help to sort out the details of these issues, as in [46]. However, for us the main point is that most of these issues concern the detailed semantics of various programming language constructions, and by interpreting generalization explicitly as a monic embedding we can sidestep these issues for the most part.

In OOP, one often defines a class A as an extension of another class B. This means that A inherits the attributes and methods of B, as if we had copied the implementation of B into that of A. Interpreting generalization with monics, we can see that this inheritance is nothing more that arrow composition. Indeed, an immutable attribute of B (say, integer-valued, for concreteness) corresponds to an arrow $B \to \texttt{Int}$ and the associated attribute on A is the composite $A \rightarrowtail B \to \texttt{Int}$.

Things are not too much trickier in the case of stateful classes. Recall that the state space of a class is defined as the product of its potential attribute values. Since every attribute of B is an attribute of A, there will be a projection from the state space of A to that of B. Flipping this around, we can model generalization in CT as a pair of maps: (i) a monic arrow between class objects and (ii) an epic arrow (projection) between state objects. Now A will inherit static attributes and pure methods through composition with the first, while inheriting state and impure methods through the second.

Before continuing, we will highlight a few points regarding this interpretation. First and most important is the observation that the generalization constructor in UML is, in some sense, redundant; essentially the same class relationship can be described in terms of a special class of decorated associations (i.e., those corresponding to monic arrows).

One might reply where UML introduces the generalization arrow, CT also introduces a new concept, the monic arrow. However, there is an important difference here. UML introduces an entirely new constructor, which must be supplied with its own semantics.

On the other hand, CT takes an existing concept, the arrow, and defines a property (monicity) which an arrow might or might not satisfy. The semantics for these monic arrows are then completely determined by our semantics for general arrows, simplifying the interpretation. Even better, because the property which defines monicity is made explicit, we can use it to make inferences in our model. The formal definition of monicity is outside the scope of this paper, but see ([19], 3.4) or ([47], 2.1 & 5.1) for an introduction.

5.3 Semantics with structures

One of the strongest arguments for the use of CT as a modeling language is its extensive collection of formal construction and properties which can be used to make the language of objects and arrows more expressive. We have already met a few of these structures in the preceding sections, most notably the Cartesian product $A \times B$ and the monic arrow $A \rightarrowtail B$ (or more precisely, the property of monicity).

Generally speaking, a construction takes some collection of objects and arrows arranged in a certain way, and uses these to create one or more new entities which reflect some relationship within the original collection. A property is similar, in that it applies to a collection of objects and arrows, but it only checks whether a condition holds rather than producing new entities.

The Cartesian product is a construction: it takes two objects $\{A, B\}$ and creates a new object $A \times B$ as well as two new arrows (the projections) $A \leftarrow A \times B \rightarrow B$. On the other hand, an arrow $f : A \rightarrow B$ is either a monic (as defined by a certain condition) or not a monic; no new pieces are added to our model, although the condition itself allows new inferences in our model.

Another important construction is called the *coproduct* or *disjoint sum*.[5] Just like the product, this construction start with two objects A and B and uses them to build a new object $A + B$. However, where the product has two projection maps out, the sum has two *in*jection pointing toward the new object.

The coproduct can be described in terms of its instances: an instance of $A + B$ is either an instance of A or an instance of B (and not both). This sounds very much like a set-theoretic union, but there is an important difference: if A and B share elements, these must be copied and placed once in either sides of the coproduct (i.e., one copy in A and one in B). This is the disjointness of the sum.

Formally the coproduct, like any CT structure, is defined purely in terms of objects and arrows. The importance of this fact, as we have already discussed with respect to monics, is that once we provide semantics for objects and arrows the interpretations of these constructions are fixed.

Like many CT constructions, the formal definition takes the form of a unique construction principle:

> *For any object Z and any two maps $f_1 : A \rightarrow Z$ and $f_2 : B \rightarrow Z$, there* exists *a* unique *map $f : A + B \rightarrow Z$ such that $f.i_1 = f_1$ and $f.i_2 = f_2$.*

CT definitions are famously abstract and unintuitive, but we can unwind the definition to see how it encodes our intuitions about disjoint unions of sets.

First, note that the equation $f.i_1 = f_1$ says that the restriction of f to A is f_1, and similarly for f_2. Based on this, we can see that the construction principle in the definition encodes disjointness while the uniqueness principle says that A and B cover the coproduct.

[5] In this discussion we will assume that our coproducts satisfy a technical condition known as extensivity. See [48] for more on this condition.

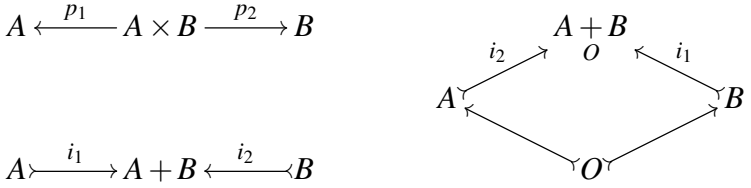


Fig. 7. The product, coproduct and pushout constructions.

For the first point, consider two functions f_1 and f_2 whose images do not overlap; if A and B had overlap, then we could not obtain both restrictions simultaneously, contrary to the existence assumption. On the other hand if there were some $x \in A + B$ belonging to neither A nor B, we could choose different images for this point without changing the restrictions to A and B, contradicting uniqueness.

In order to model the ordinary set-theoretic union in CT we need a more sophisticated construction called the *pushout* and denoted $A \underset{O}{+} B$. In this case the construction begins with additional information: we must explicitly represent the overlap of A and B as a separate class O, together with two maps $A \hookleftarrow O \hookrightarrow A$; these situate O inside A and B. Based on this data the construction produces one new object and two new injections, just like the coproduct construction. The difference here is that whereas the coproduct is disjoint, the pushout overlaps at O.

Now we will compare the CT approach via coproducts and pushouts against a similar UML construction called a *generalization sets*. We indicate generalization sets in a diagram by joining several generalization arrows into a single tree. For comparison, the subclasses displayed in figure 6 are separate generalizations, whereas the same classes are shown in figure 1 as a generalization set.

Formally, a generalization set is just what it sounds like: a set of generalizations. In the two figures just mentioned, there is no difference between the two different representations. However, we can extend the semantics of a generalization set by decorating it with two binary tags: complete/incomplete and disjoint/overlapping. The first determines whether the collection of subclasses covers the full range of the super class, and the second identifies whether the subclasses may intersect one another. If the tags are omitted, these properties default to incomplete and overlapping.

In CT, we have already seen that coproducts and pushouts allow us to express disjointness and overlap. We can combine these with other tools, epics and monics, to express facts about coverage. This allows us to translate the UML decorations on generalization sets into different, visually distinct CT models.

Moreover, when we translate into CT we can see that some models are simpler than others. In particular, the "incomplete" options is a bit awkward to model: we must introduce a new "remainder" object R to fill out the rest of the superclass. Similarly, "overlapping" models are more complicated because they require the explicit representation of an overlap class.

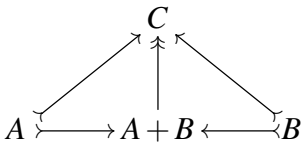
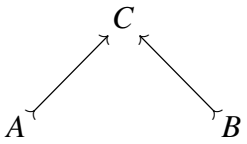
Covering Type	Intersection Type: Disjoint	Intersection Type: Agnostic	Intersection Type: Overlapping
Complete	$A+B \cong C$		$A \underset{O}{+} B \cong C$
Agnostic	$A+B \rightarrowtail C$		$A \underset{O}{+} B \rightarrowtail C$
Incomplete	$A+B+R \cong C$ $R \ncong \emptyset$	$A+R \rightarrowtail C \leftarrowtail B+R$ $R \ncong \emptyset$	$\left(A \underset{O}{+} B\right)+R \cong C$ $R \ncong \emptyset$

Table 1. CT interpretations for different types of UML generalization sets.

In fact, this awkwardness arises from these UML decorations have omitted an important possibility: we may not know whether two classes overlap or cover! CT models easily support this agnostic point of view, and the resulting models are simpler than those associated with the UML's default options. All nine possible models are included in table 1.

We have only scratched the surface of CT constructions here. Other important examples include equalizers, which encode the extraction of solution sets, and pullbacks, which model database joins. Just as importantly, these constructions are embedded in a mathematical theory which details their interrelationships. For example, a simple CT lemma says that the pullback of a monic arrow is again monic; this and other facts allow us to make automatic inferences about our model based on how it is put together.

There is also a wide literature relating various categorical structures to other areas of mathematics such as formal logic and the theory of computation. From this we know, for example, that the expressive capability of CT varies depending on which constructors we choose to employ. Products allow us to model equality and conjunction (algebraic logic); image factorization based on epics and monics allows for existential quantification; implication and universal quantification require more advanced tools like the function spaces we will meet in the next section. We can keep going... categories known as toposes support full higher-order logic and type theory.

Thus, not only is CT an extremely expressive modeling language, we can also modulate that expressivity based on what features we need to model. This is important because the expressivity is inversely proportional to analyzability: it is usually easier to verify or prove statements written in weaker logics. In CT we can match this level to our problem,

simplifying these analyses. Moreover, more sophisticated CT constructions called adjunctions act as partial dictionaries between different logics, specifying whether and how we can transport our models from one modeling context to another.

6. Other class diagram relationships

Now we turn to some constructions in the UML class diagram whose semantics are less clear for one reason or another. Some of these can be supported in CT, though at the cost of greater complexity. Others, we argue, try to smuggle activity information into the class diagram which doesn't really belong there. Further problems arise from UML's entanglement with software development, and may not be appropriate in a truly unified modeling language. Overall, we reason that the added complexity needed to deal with these constructions is not worth the effort, and they could be removed without a substantial loss expressive power in describing class relationships, though some elements of these constructions may be relevant for modeling *processes*, the topic for our next paper in this series.

6.1 Aggregation & Composition

Aggregation and composition in UML class diagrams are special types of associations, indicated by diamond heads as in figure 8:

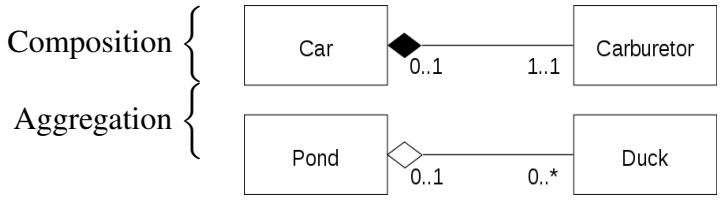


Fig. 8. Example aggregation and composition associations in UML [2].

Both are intended to express some sort of part/whole relation, but composition is intrinsic while aggregation is potentially transient. This is often explained by saying that in aggregation you can remove the whole and still have its parts, whereas in composition removing the whole also destroys its parts.

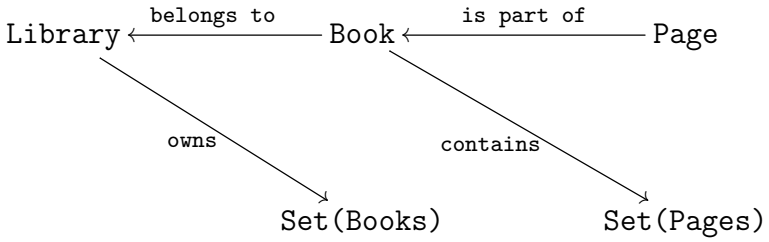
These constructions suffer from several of the complaints listed above. The origin of the distinction is in object-oriented software, where these considerations were important for memory management. The fact that the difference between the two is described in operational terms (removal) suggests that there is some component of activity being projected onto the class relationship. Most importantly, it seems that this distinction is context dependent. The relationship between my car and its tires should act like composition when I am driving, but like aggregation when the car is in the shop. Context sensitivity is another indication that something beyond class relationships is being expressed here. Given all this,

we propose giving up on the distinction between these two constructs, at least at the level of class diagrams.

Nonetheless, aggregation and composition do express an important concept, which we think of at some times as ownership and at others as internal structure. For example, every library owns a set of books, while every book contains a set of pages. Structurally, these two relationships are quite similar, associating an instance of one class with a set of instances from another. There is a categorical construction called a dependent type[6] or a fibration which expresses a similar idea.

Roughly speaking, a dependent type is a set whose definition involves one or more (typed) variables. A typical example in mathematics is the vector space $\mathbb{R}^n$, which depends on a natural-number variable $n : \mathbb{N}$; we can usually think of a dependent type as a family of sets $\{B_a\}_{a \in A}$, where $a \in A$ ranges over the indexing variable(s) of the dependent type. The key observation here is that we can always wrap up a set of sets into a single function, so that, to a first approximation, a dependent type is nothing more than an arrow "thought of in reverse".

Usually we think of a mapping $f : B \to A$ as a map which sends each element $b \in B$ to a unique element $a = f(b) \in A$. However, we can turn this around and think of it as a map which takes an element $a \in A$ and returns *a set of* elements $B_a \subseteq B$. Specifically, B_a is the set of element $b \in B$ such that $f(b) = a$, often called the *fiber* of B over a. This idea allows us to encode the examples mentioned above:



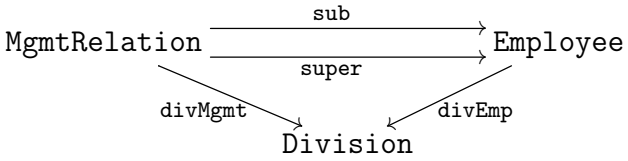
Single maps are sufficient to encode sets of sets, but commutative triangles are required to express additional relationships between these. For example, a directed graph can be represented by a pair of objects (nodes and edges) and two arrows (source and target) mapping edges to nodes. This corresponds to a small category

$$\texttt{Edges} \underset{t}{\overset{s}{\rightrightarrows}} \texttt{Nodes}\,.$$

Now suppose that we would like to model a company in which each division has a management structure described by a directed graph, with employees as nodes and supervisory relations as edges. We can represent this categorically by fibering the graph category over

[6] This is a *different* notion of dependency than that found in UML, which we will discuss below.

the class of divisions:



$$\texttt{super.divEmp} = \texttt{divMgmt} = \texttt{sub.divEmp}$$

This specifies that each division has, as part of its internal structure, a directed graph whose nodes are employees and whose edges are management relations. The equations specify that supervisors and subordinates belong to the same division (ownership), as does the management relation between them. Notice that this specification already enforces some of the inferences attributed to composition. If we remove one of these divisions without changing anything else, we are left in an inconsistent state: the `depEmp` map is no longer functional because some employees are missing a division. To recover consistency we must either remove or redirect them. Which of these is appropriate is a question of activities rather than class relationships.

We can also use the dependent type construction to model some additional numeric constraints which we passed over in section 5 (see footnote 4).

Suppose we want to say that every A has exactly two B's. It follows that B is isomorphic to a coproduct $B \cong A+A$, but this requirement alone is not enough: it only says that every A has two B's *on average*. To impose the constraint fiber-by-fiber we note that the coproduct has a canonical map (the *codiagonal*) $\nabla : A+A \to A$, and require the isomorphism commutes over A. The same construction works for other multiplicities and, more generally, we can model upper or lower bounds rather than exact multiplicities by using monic arrows rather than isomorphisms, as shown in Figure 9.

In closing we note one further important but subtle point. In the context of sets and functions, there really is no difference between functions and dependent types. However, in other semantic contexts, especially those which involve some notions of continuity, the two context diverge, and dependent types are a more restrictive concept. These restrictions are usually based on the ability to lift structure in the base (such as parthood or substitutability) to corresponding relationships in the total space.

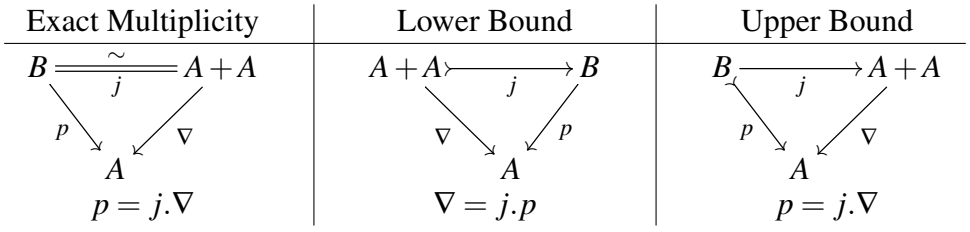


Fig. 9. Multiplicity constraints as dependent arrows.

In most cases, though, we can simply use this method to embed whole/part relations into our class diagrams. If we feel that it is useful we can add a special notation for arrows which express ownership or structure in this way; mathematicians sometimes use a slashed arrow $B \twoheadrightarrow A$ to indicate dependent types. Although this construction supports categorical semantics stronger than simple association, its main benefit here is conceptual: declaring ownership or internal structure helps us to understand the intention of the model. However, the distinction is invisible at the level of sets and functions, and so rarely enters directly into our considerations.

6.2 Dependency

The next diagram construction we consider is UML dependency, which is a different notion than the categorical dependency discussed in the previous section, though there is a relationship between the two. In UML, dependency is expressed by a dashed arrow like the one in figure 10.

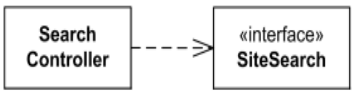


Fig. 10. Dependency relation in UML [3]

Dependency is another construction which has its roots in software development, where it is usually interpreted in terms of `import` statements, allowing one class access to the methods of another. Officially, the UML construction is more general, applying also to abstractions such as the relationship between a specification and an implementation. Unfortunately, this generality is achieved through weakening the semantics of dependency to the point that it is little more than an annotation attached to an association.

This means that we won't lose much if we remove dependency from the language. Rather than having a special constructor for implementation/abstraction, we can just create an ordinary association `Implements` to capture this relationship. If there are special features of abstraction and implementation (e.g., validation and verification requirements), these should be explicitly called out as attributes of the new relation object. Without providing more meaningful descriptions of these dependencies, the extra diagram component provides intuition but little more.

As we think through the details of these descriptions, we see that the different relationships UML bundles together as dependencies may have little in common. Abstraction relationships are usefully modeled as class associations. On the other hand, usage-based dependencies like those created by `import` statements usually have more to do with the methods of a class than with its instance.

In order this type of dependence through at CT lens we need another construction, called the exponential. Given two classes X and Y we can consider the set of (computable)

functions $Y \to X$, a new class denoted X^Y. When X and Y have n and k elements, respectively, there are n^k such functions, justifying the exponential notation.

The most important operation associated with exponentials is called currying. Given a binary function of X and Y taking values in Z, we can curry to represent this instead as a unary function of X, taking values in the exponential Z^Y. Diagrammatically, the two arrows below encode equivalent information:

$$\frac{X \times Y \xrightarrow{\quad b \quad} Z}{X \xrightarrow{\quad \texttt{curry}(b) \quad} Z^Y}$$

Given the curried arrow, we can recover the original binary function by composing with the evaluation map $\texttt{ev} : Z^Y \times Y \to Z$.

For example, suppose that we have a search engine implemented in one class which scores websites based on a scoring algorithm implemented in another class. Typically, these two classes cannot interact because they inhabit different scopes; dependency brings the scoring algorithm into the scope of the search engine.

Suppose, for simplicity, that the search engine only returns the top document for a query, so that we can represent the algorithm as an arrow $\texttt{search} : \texttt{Query} \to \texttt{Doc}$. This relies on a scoring algorithm $\texttt{score} : \texttt{Doc} \times \texttt{Query} \to \mathbb{N}$. The search simply applies the score map to each document in a corpus, sorts the results and returns the top scorer.

In particular, this means that when we change our scoring algorithm, even if the `search` code is the same, we will get different search results. This means that the code in `search` does *not* define a function $\texttt{Query} \to \texttt{Doc}$ (in the mathematical sense). Implicitly, it contains another variable.

It is instructive to think about what would happen if we tried to run the `search` code in isolation: the compiler would complain that the `score` variable is undefined. It is more accurate to think of `search` as a binary function, whose (implicit) second variable is of exponential type:

$$\texttt{search} : \texttt{Query} \times \mathbb{N}^{\texttt{Doc} \times \texttt{Query}} \longrightarrow \texttt{Doc}.$$

This is usually what we mean when we say that one method depends on another: the first implicitly employs the the second in the calculation, and this can be made explicit as an exponential argument. One can easily generalize this to multiple dependencies. With this we can concretely describe class dependencies: class X depends on class Y if X contain methods which implicitly refer to function variables which are implemented in Y.

Notice that this necessitates a substantial change in perspective. Whereas we typically identify a class with its set of instances, dependency associates them with the methods which they implement. Again, it is not clear that this information belongs in a diagram of class relations, especially if we are talking about real-world entities rather than object-oriented programming.

6.3 Navigation

Navigation is the last decoration which can be attached to an assocation; an arrow head indicates that one direction of an association is navigable, while an $\times$ across the arrow means that it is not. For example, a customer should be able to see the items on their order, but we may want to hide information about which customers purchase a particular item.

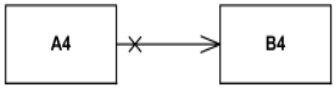


Fig. 11. Navigation relations in UML [4].

Again, we see operational descriptions for this UML component; navigation specifies information about user and component interaction, specifically in regards to information flow. This is undoubtedly important information, but may not be relevant to class relationships. Better to model such issues properly in some sort of activity diagram.

Even though we argue for removing navigation considerations from class diagrams, we can still gain some insight into the issue by considering the situation from a categorical perspective. The key fact is that we can *always* think of an association (i.e., a relation) in four equivalent ways. Each of these four corresponds to a different notion of information flow.

Take the association between students and classes in a university. In one direction, we can map each student to their schedule, a set of classes. Going the other way, we can map each class to its roster, a set of students. The same information can also be represented as a subset of pairs satisfying the association, or as a binary truth function. These correspond to four different (but equivalent) arrows, where $\texttt{Set}(X)$ represents the set of subsets of X.

$$\texttt{Student} \longrightarrow \texttt{Set}(\texttt{Class})$$

$$\texttt{Class} \longrightarrow \texttt{Set}(\texttt{Student})$$

$$\texttt{Student} \times \texttt{Class} \longrightarrow \{\texttt{T},\texttt{F}\}$$

$$\texttt{Enrollment} \rightarrowtail \texttt{Student} \times \texttt{Class}$$

The key point is that these maps exist for any association; we can't create or eliminate the channels by drawing arrowheads and $\times$'s. The decision on when and whether to grant access to these channels has more to do with activity than with class relationships, and so belongs elsewhere.

7. Comparing models

In the last few sections we have walked through the main constructions in the UML class diagram, and discussed how these can be interpreted in terms of categorical terms. Now we will briefly compare our initial example of a UML diagram (figure 1) with the CT model derived from it. Both models are shown in figure 12.

In this CT diagram we have suppressed all but the most important arrows associated with attributes and methods; we use unboxed objects to indicate data types. This choice cuts down on clutter and emphasizes the similarity between the two models. It is a simple matter to add these arrows in, based on the procedure given in section 4. Alternatively, these pieces of the model could be included in separate diagrams; a later paper in this series will discuss the use of CT constructions called colimits to build up aggregate models from modular pieces. We are also free to use a notation analogous to UML, listing the names and types of the attributes and methods, so long as we understand that this is nothing more than a syntactic sugar, which can be unambiguously translated into a diagram of objects and arrows.

The first thing to notice about these two diagrams is their essential similarity: the two diagrams have the same basic shape (except for the attributes included in the CT model), and it is easy to identify a dictionary between the two models. In part, this similarity is over-stated because all of the associations included in the original model were *already* functional; we did not need to create any new vertex objects to mediate these relations.

In fact, the UML model could have been presented differently. It seems reasonable that different OrderDetails for the same Order will refer to different Items; this means that OrderDetails is a relation between Order and Item, and could have been omitted in favor of a direct association. In UML seeing this requires parsing the multiplicity decorations of both associations, whereas in the CT model it is immediately obvious from the directions of the two arrows.

Another important observation is that the CT diagram strips out many of the semantically fuzzy features of the UML diagram. Sometimes these are replaces by analogous CT features but, critically, all of these CT constructions are defined in terms of simple objects and arrows, so their interpretation is fixed from the outset.

For example, we have replaced the UML generalization set with a coproduct of classes. In fact, this is *not* an accurate translation of the UML diagram; an unlabelled generalization set is assumed to be overlapping and incomplete. However, our knowledge of the domain suggests that these three classes should definitely be disjoint and probably exhaustive, suggesting that the diagram's authors merely forgot to decorate the construction. Based on the discussion in section 5.3 it would be easy to modify the model based on other choices, or to leave these decisions unmade for now.

We can also do away with the notion of an abstract class. In our CT model, `Payment` is just another class, not fundamentally different from any other. The special feature in CT which corresponds to its abstract character in UML is that the class can be *defined* as a coproduct of other, smaller classes. We suspect that this is a more general fact: abstract

classes in UML correspond to colimit objects (i.e., coproduct, pushout, etc.) in CT.

The CT model also forces us to be more specific in places. Most of the attributes and methods listed in the UML diagram are not given types, although in most cases we can infer the type based on its name. In CT, we have no choice but to make these inferences explicit. Quantities and amounts are numbers, whereas dates, even if they are given numerically, are not just numbers.

Forcing this information also helps to remove ambiguity from our models. While most of the attributes in the original diagram are fairly intuitive, some are not. What does the Cash attribute of the Cash class mean? If we knew what type of entity it was, this would help, even though it would not pin down the answer to the question.

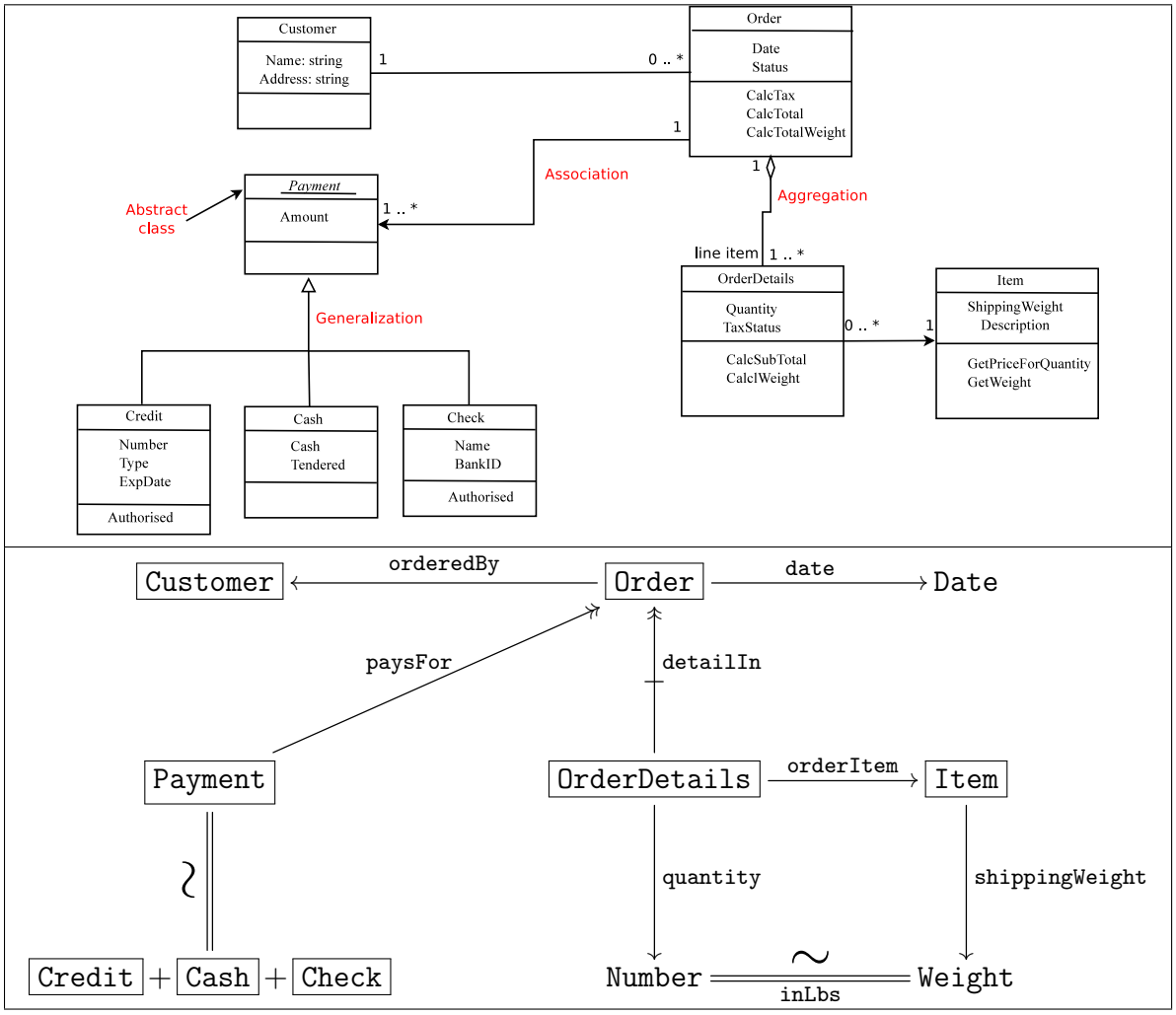


Fig. 12. An example UML class diagram [1] and its translation into CT. Most attributes and methods have been suppressed to avoid clutter.

On a related note, the use of isomorphism in CT provides a useful method for distinguishing between equivalent types. For example, the `quantity` of an `OrderDetail` and the `shippingWeight` of an `Item` can both be expressed numerically, but they are not the same type of numbers: the first is a unitless number, whereas the second is a physical quantity which does not have a numeric value *until we fix a unit of measure*.

By representing these as separate objects, we are forced to add in an explicit isomorphism which encodes this choice of units. This allows us to assert that some combinations of numbers make sense while others do not. We cannot add a unitless number to a weight, just as we cannot add a length to a time interval. On the other hand, it does make sense to scale (i.e., multiply) a physical quantity by a unitless number, a fact which we make use of below.

Although we have not pursued the issue here, we can also use CT in order to specify the meaning of certain attributes or methods more directly, using path equations. For example, the total weight of an Order (the CalcTotalWeight method) should be calculated starting from the set of its OrderDetails. For each detail, we can calculate its weight (the CalcWeight method), say in pounds, based on the quantity and item. Summing these up gives the total weight.

In CT, both operations can be defined in terms of a chain of arrow compositions, which can be derived from the mathematical formula for each method. These are shown in figure 13. The arrow `Set(calcWeight)` which appears in the second method requires some explanation. For any map $f : A \to B$ we can define a related map $\texttt{Set}(f) : \texttt{Set}(A) \to \texttt{Set(B)}$, which takes an input set $\{a_1, a_2, \ldots, a_n\}$ and applies f to its elements one by one, returning the set $\{f(a_1), \ldots, f(a_n)\}$. This reflects the fact that the `Set` construction is an example of a categorical operation called a functor, a concept which we will explore later in this series of papers. Notice, also, that the calculation of total weight in the `Order` class requires access to methods and attributes from `OrderDetails` and `Item`, suggesting that the UML model might be missing dependency relations between these classes. In CT, we remove explicit dependency from our model, but it is nonetheless obvious from the definitions of these methods. Moreover, this provides more information about *which* methods in a class are dependent, and precisely what they depend on.

8. Conclusion

This is the first in a series of papers analyzing the Unified Modeling Language (UML), and proposing an alternative approach to modeling based on the mathematical theory of categories. Here we focussed on the UML class diagram, leaving other aspects of the language for future work.

Our criticisms of UML fall into several dimensions. Perhaps foremost is the lack of formal semantics for UML as a modeling language. This can lead to ambiguity in UML models, where certain constructions may be interpreted in different ways by different people, limiting communication of our models both between humans and with computing systems. This problem is compounded by UML's tendency to introduce new constructions into the

language, rather than modeling new ideas through designs in the existing language. This adds to the semantic issues by adding new terms in need of interpretation. It also makes the language harder to learn and to use, because there may be many different ways to model the same core concept, with little indication of how these choices differentiate. Finally, the universality of UML is limited by its origins in object-oriented programming. This introduces features into the language which may not be appropriate in other domains. It also discourages a separation of concerns, as active concepts like navigation sneak their way into models meant to contain static relationships.

Categorical modeling avoids all these concerns. CT is already a mathematical language, so there is no concern for ambiguity or informality in our models. Moreover, the semantics of categorical models is a well-studied area of mathematics, yielding connections not only with formal logic and computation but also other areas of mathematics like probability and dynamical systems. Despite connections to so many formal domains, the field's abstraction ensures that the modeling language itself is domain agnostic. Moreover, CT's assumptions are minimal–objects and arrows–with the rest of the theory defined through design patterns rather than declared by fiat. Best of all, we can do all this without losing the diagrammatic charater of UML which is so important for intuitive design and communication.

The next paper in this series will be similar to the present work, translating UML *activity* diagrams into categorical terms using a formal syntax called *string diagrams*. With these two tools in hand, we will then explore the relation between these, showing how we can link these methods together as two viewpoints on a single, unified model encompassing both static and active relationships. As a demonstration of the utility of this approach, we will show that in CT the other facets of UML–the use case diagram, the interaction diagram, the object diagram, etc.–can be decompiled into activity and class representations, obviating the need for all these different diagrams except, possibly, as a syntactic sugar for the true models. We will also begin to explore the ways that CT goes beyond UML; in par-

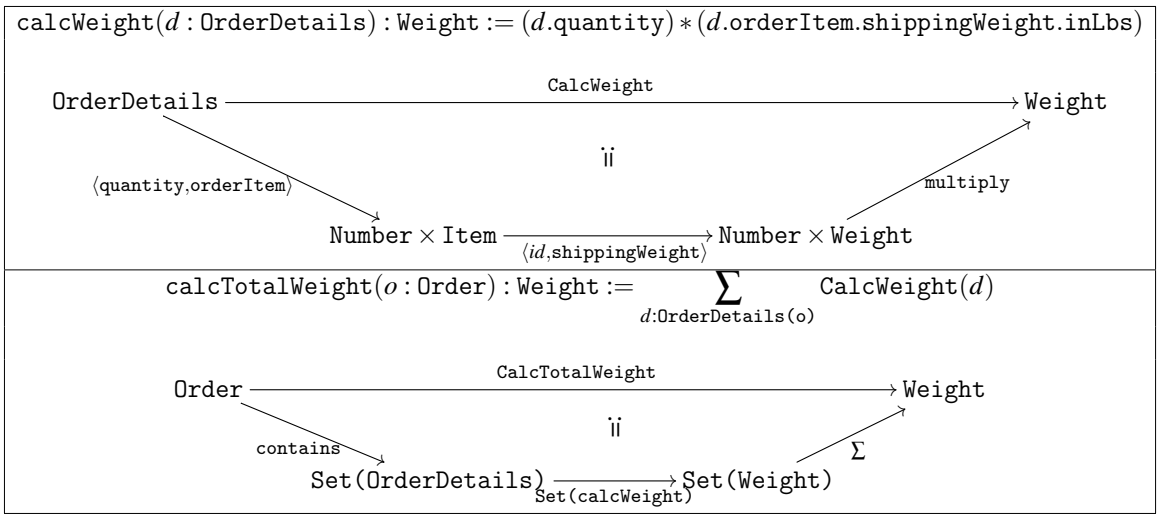


Fig. 13. Defining methods via composition of arrows.

ticular, we will examine the use of functors (mappings between categories) and other CT constructions for practical model management, including model transformation, integration and evolution.

Applied category theory is a field still in its infancy, though progressing fast. One of the central obstacles to its growth is the lack of a standard methodology for applying ideas from CT in concrete contexts. This paper begin to develop such a methodology by adopting the successes of UML while avoiding its pitfalls. Eventually, we hope to develop this work into a more formal Categorical Modeling Language (CatML), along with a broad collection of design patterns for modeling various specific circumstances.

9. Acknowledgements

We would like to acknowledge the range of valuable comments and suggestions from Blake Pollard, Jonathan Patrick and Ken Baclawski that made the paper better. However, we are responsible for any errors or mistakes.

References

[1] Booch G, Rumbaugh J, Jacobson I (2005) *Unified Modeling Language User Guide, 2nd Edition*. Addison-Wesley Object Technology Series (Addison-Wesley Professional), .

[2] Wikipedia (2017) Class diagram — wikipedia, the free encyclopedia, https://en.wikipedia.org/w/index.php?title=Class_diagram&oldid=805715397. Online; accessed 19-October-2017.

[3] uml-diagramsorg (2017) Dependency in UML — uml-diagrams.org, http://www.uml-diagrams.org/dependency.html. Online; accessed 19-October-2017.

[4] uml-diagramsorg (2017) UML class diagrams reference — uml-diagrams.org, http://www.uml-diagrams.org/class-reference.html. Online; accessed 19-October-2017.

[5] Frigg R, Hartmann S (2006) Models in science, https://seop.illc.uva.nl/entries/models-science/. Accessed: 2018-66-20.

[6] Giere RN (2004) How models are used to represent reality. *Philosophy of Science* 71(5):742–752.

[7] Robinson S (2010) Conceptual modelling: Who needs it. *SCS M&S Magazine* 2:1–7.

[8] Robinson S (2011) Choosing the right model: Conceptual modeling for simulation. *Simulation Conference (WSC), Proceedings of the 2011 Winter* (IEEE), , pp 1423–1435.

[9] Robinson S (2013) Conceptual modeling for simulation. *Simulation Conference (WSC), 2013 Winter* (IEEE), , pp 377–388.

[10] Kühne T (2005) What is a model? *Language Engineering for Model-Driven Software Development*, eds Bezivin J, Heckel R number 04101 in Dagstuhl Seminar Proceedings (Internationales Begegnungs- und Forschungszentrum für Informatik

(IBFI), Schloss Dagstuhl, Germany, Dagstuhl, Germany), , pp 1–10. Available at http://drops.dagstuhl.de/opus/volltexte/2005/23.
[11] Muller PA, Fondement F, Baudry B, Combemale B (2012) Modeling modeling modeling. *Software & Systems Modeling* 11(3):347–359.
[12] Rumbaugh J, Blaha M, Premerlani W, Eddy F, Lorensen WE, et al. (1991) *Object-Oriented Modeling and Design*. Vol. 199.1 (Prentice-hall Englewood Cliffs, NJ), .
[13] Petre M (2013) UML in practice. *Proceedings of the 2013 International Conference on Software Engineering* (IEEE Press), , pp 722–731.
[14] Reggio G, Leotta M, Ricca F (2014) Who knows/uses what of the uml: A personal opinion survey. *International Conference on Model Driven Engineering Languages and Systems* (Springer), , pp 149–165.
[15] Object Managment Group (2017) Unified Modelling Language version 2.5. Unified Modelling.
[16] Evans A, France R, Lano K, Rumpe B (1999) Meta-modelling semantics of UML. *Behavioral Specifications of Businesses and Systems* (Springer), , pp 45–60.
[17] Diskin Z (2002) Visualization vs. specification in diagrammatic notations: A case study with the UML. *Diagrammatic Representation and Inference* :79–127.
[18] Diskin Z (2003) Mathematics of UML: Making the odysseys of UML less dramatic. *Practical Foundations of Business System Specifications* (Springer), , pp 145–178.
[19] Spivak DI (2014) *Category Theory for the Sciences* (The MIT Press), .
[20] MacLane S, Moerdijk I (2012) *Sheaves in Geometry and Logic: A First Introduction to Topos Theory* (Springer Science & Business Media), .
[21] Spivak DI, Kent RE (2012) Ologs: a categorical framework for knowledge representation. *PLoS One* 7(1):e24274.
[22] Eilenberg S, MacLane S (1945) General theory of natural equivalences. *Transactions of the American Mathematical Society* 58(2):231–294.
[23] Hartshorne R (1977) *Algebraic Geometry* (Springer-Verlag, New York-Heidelberg), . Graduate Texts in Mathematics, No. 52.
[24] Hatcher A (2000) *Algebraic Topology* (Cambridge Univ. Press, Cambridge), .
[25] Lawvere FW (1963) Functorial semantics of algebraic theories. *Proceedings of the National Academy of Sciences* 50(5):869–872.
[26] Lawvere FW (1970) Quantifiers and sheaves. *Actes du Congres International des Mathematiciens, Nice*, Vol. 1, pp 329–334.
[27] Lipovaca M (2011) *Learn You a Haskell for Great Good!: A Beginner's Guide* (no starch press), .
[28] Barr M, Wells C (eds) (1995) *Category Theory for Computing Science, 2nd Ed.* (Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK), .
[29] Penrose R (1971) Applications of negative dimensional tensors. *Combinatorial Mathematics and its Applications* 221244.
[30] Selinger P (2010) A survey of graphical languages for monoidal categories. *New Structures for Physics* (Springer), , pp 289–355.
[31] Culbertson J, Sturtz K (2013) Bayesian machine learning via category theory. *arXiv*

preprint arXiv:13121445 .

[32] Culbertson J, Sturtz K (2014) A categorical foundation for bayesian probability. *Applied Categorical Structures* 22(4):647–662.

[33] Lerman E (2016) A category of hybrid systems. *ArXiv e-prints* 1612.01950.

[34] Vagner D, Spivak DI, Lerman E (2015) Algebras of open dynamical systems on the operad of wiring diagrams. *Theory and Applications of Categories* 30(51):1793–1822.

[35] Johnson M, Rosebrugh R, Wood R (2002) Entity-relationship-attribute designs and sketches. *Theory and Applications of Categories* 10(3):94–112.

[36] Rosebrugh R, Wood R (1992) Relational databases and indexed categories. *Proceedings of the International Category Theory Meeting 1991, CMS Conference Proceedings*, Vol. 13, pp 391–407.

[37] Spivak DI (2012) Functorial data migration. *Information and Computation* 217:31–51.

[38] Healy MJ, Caudell TP (2006) Ontologies and worlds in category theory: Implications for neural systems. *Axiomathes* 16(1-2):165–214.

[39] Ehresmann AC, Vanbremeersch JP (2007) *Memory Evolutive Systems; Hierarchy, Emergence, Cognition* (Elsevier), .

[40] Pollard BS (2016) Open markov processes: A compositional perspective on non-equilibrium steady states in biology. *Entropy* 18(4):140.

[41] Baez JC, Fong B, Pollard BS (2016) A compositional framework for markov processes. *Journal of Mathematical Physics* 57(3):033301.

[42] Patterson E (2017) Knowledge representation in bicategories of relations. *arXiv preprint arXiv:170600526* .

[43] Cabot J, Gogolla M (2012) Object constraint language (ocl): a definitive guide. *Formal Methods for Model-Driven Engineering* (Springer), , pp 58–90.

[44] Spivak DI, Wisnesky R (2015) Relational foundations for functorial data migration. *Proceedings of the 15th Symposium on Database Programming Languages* (ACM), , pp 21–28.

[45] Baclawski K (1990) The structural semantics of subtypes and inheritance. Accessed: 6/20/2018.

[46] Reynolds JC (1980) Using category theory to design implicit conversions and generic operators. *International Workshop on Semantics-Directed Compiler Generation* (Springer), , pp 211–258.

[47] Awodey S (2010) *Category Theory* (Oxford University Press), .

[48] nLab (2016) Extensive category — nLab, https://ncatlab.org/nlab/show/extensive+category. Online; accessed 19-October-2017.