An introduction to Category Theory for Software Engineers*

Dr Steve Easterbrook

Associate Professor,

Dept of Computer Science,

University of Toronto

sme@cs.toronto.edu

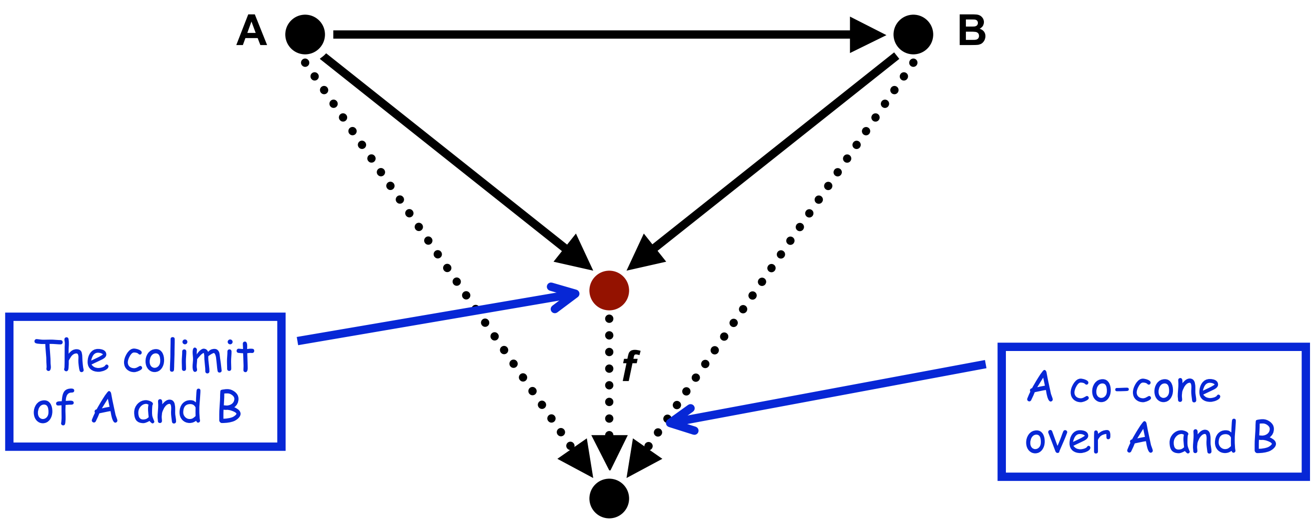
*slides available at http://www.cs.toronto.edu/~sme/presentations/cat101.pdf

Key Questions for this tutorial

- What is Category Theory?
- Why should we be interested in Category Theory?
- How much Category Theory is it useful to know?
- What kinds of things can you do with Category Theory in Software Engineering?
- *(for the ASE audience)*
  Does Category Theory help us to automate things?

By way of introduction...

- **An explanation of "Colimits"**



- **My frustration:**
  - → Reading a maths books (especially category theory books!) is like reading a program without any of the supporting documentation. There's lots of definitions, lemmas, proofs, and so on, but no indication of what it's all for, or why it's written the way it is.
  - → This also applies to many software engineering papers that explore formal foundations.

Outline

(1) An introduction to categories

- Definitions
- Some simple examples

you
are
here

(2) Motivations

- Why is category theory so useful in mathematics?
- Why is category theory relevant to software engineering?

(3) Enough category theory to get by

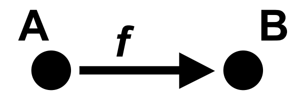
- some important universal mapping properties
- constructiveness and completeness

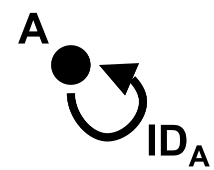
(4) Applying category theory to specifications

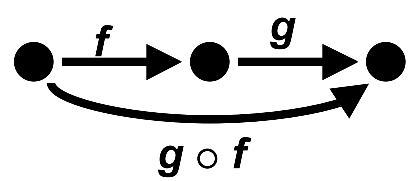
- Specification morphisms
- Modular Specifications
- Tools based on category theory

Definition of a Category

- **A category consists of:**
  - → a class of *objects*
  - → a class of *morphisms* ("arrows")
  - → for each morphism, f, one object as the *domain* of f and one object as the *codomain* of f.
  - → for each object, A, an *identity morphism* which has domain A and codomain A. ("ID_A")
  - → for each pair of morphisms $f:A\to B$ and $g:B\to C$, (i.e. $cod(f)=dom(g)$), a *composite morphism*, $g \circ f: A\to C$



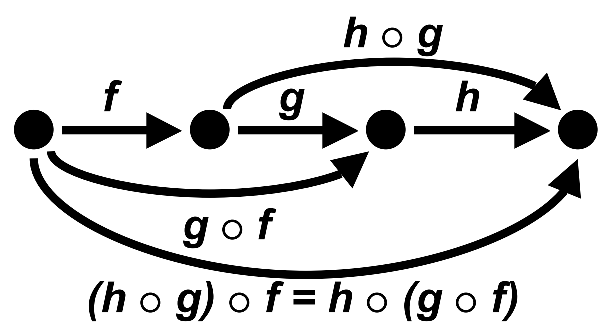




- **With these rules:**
  - → *Identity composition:* For each morphism $f:A\to B$,

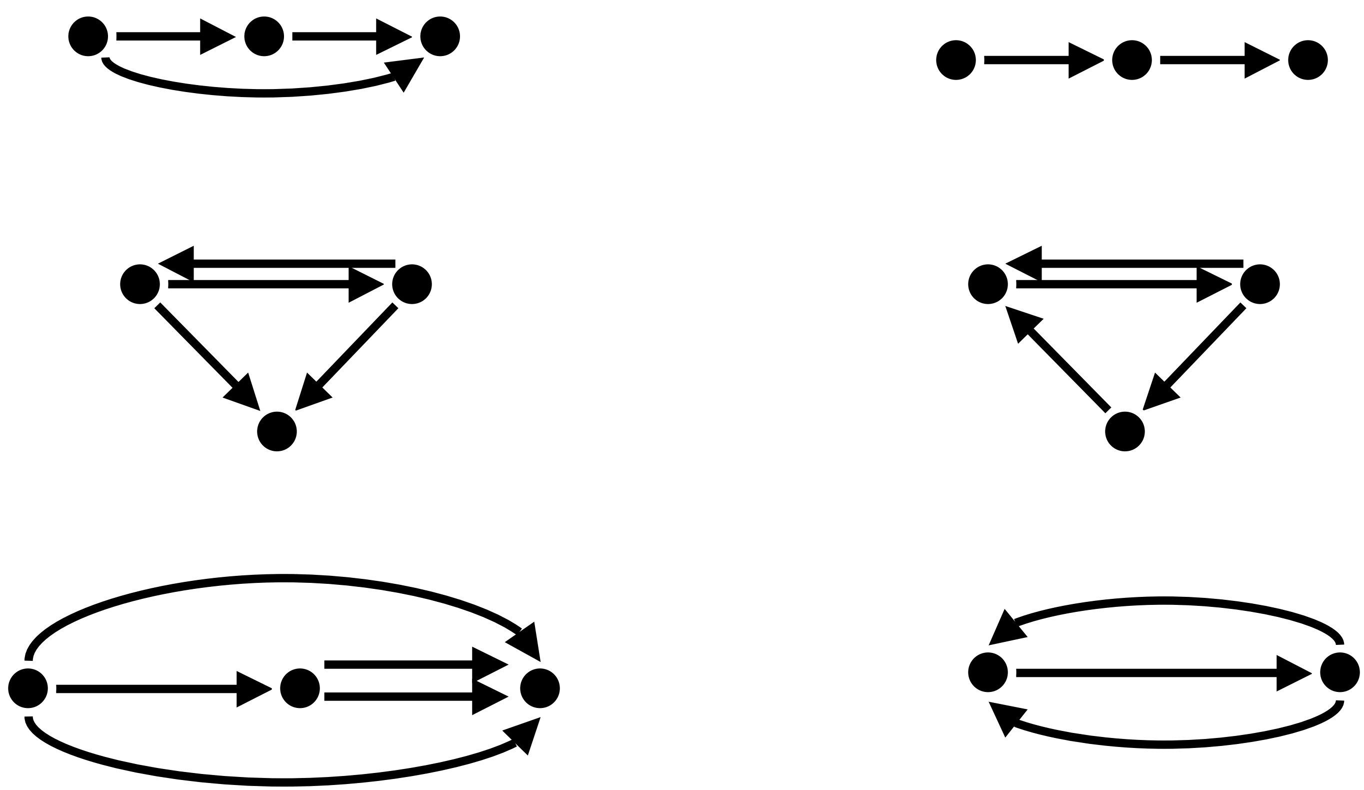
    $f \circ ID_A = f$ and $ID_B \circ f = f$
  - → *Associativity:* For each set of morphisms $f:A\to B$, $g:B\to C$, $h:C\to D$,

    $(h \circ g) \circ f = h \circ (g \circ f)$



Understanding the definition

Which of these can be valid categories?



Note: In this notation, the identity morphisms are assumed.

Understanding the definition

Proof that $\begin{tikzcd} \bullet \arrow[r] & \bullet \arrow[l, bend right] \arrow[l, bend left] \end{tikzcd}$ **is not a category:**

$$\begin{tikzcd} A \arrow[loop left, "ID_A"] \arrow[r, "f"] & B \arrow[loop right, "ID_B"] \arrow[l, bend right, "g"'] \arrow[l, bend left, "h"] \end{tikzcd}$$

Composition:

$$f \circ h = \mathrm{ID_B}$$
$$f \circ g = \mathrm{ID_B}$$
$$h \circ f = \mathrm{ID_A}$$
$$g \circ f = \mathrm{ID_A}$$

✔

okay so far

Associativity:

$$h \circ f \circ g = (h \circ f) \circ g = \mathrm{ID_A} \circ g = g$$
$$h \circ f \circ g = h \circ (f \circ g) = h \circ \mathrm{ID_B} = h$$

Hence: $g = h$

not okay

Note: $h \circ f = g \circ f \not\Rightarrow h = g$, although it may in some categories.

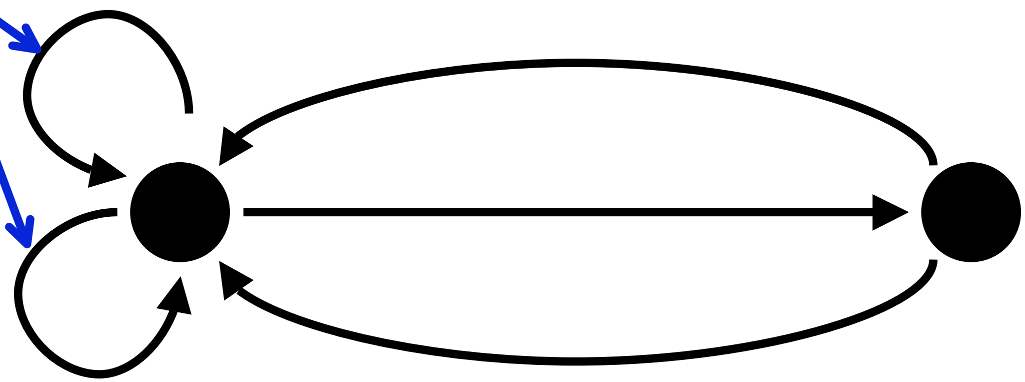
Hence, $\begin{tikzcd} \bullet \arrow[r] \arrow[rr, bend left] & \bullet \arrow[r, shift left] \arrow[r, shift right] & \bullet \end{tikzcd}$ can be a category.

Challenge Question

(For the experts only)

Can this be a category?

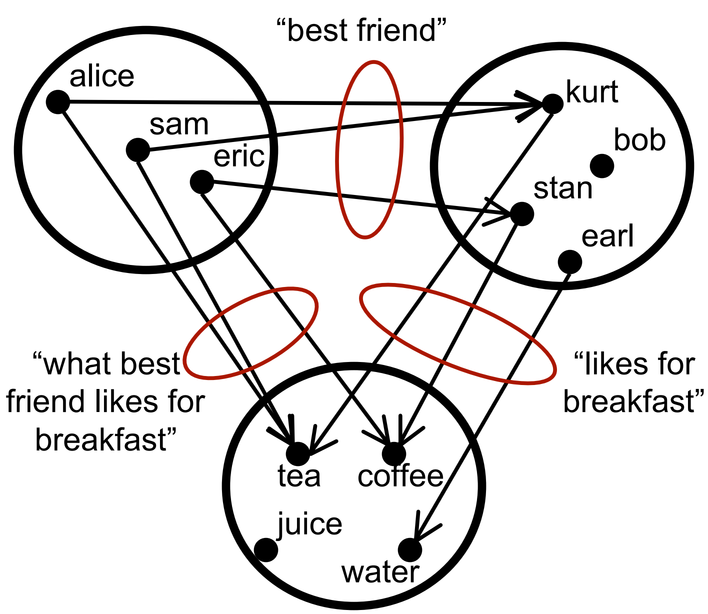
These are not identities



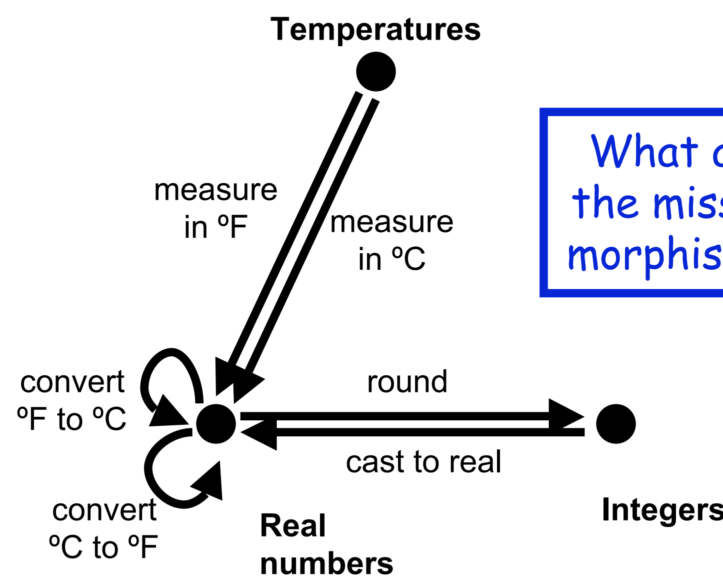
Example category 1

- **The category of sets** *(actually, "functions on sets")*
  - → objects are sets
  - → morphisms are functions between sets

E.g.



E.g.



What are the missing morphisms?

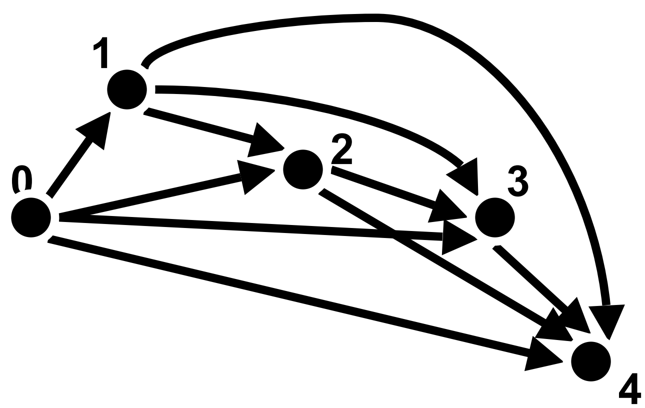
Example category 2

- **Any partial order (P, ≤)**
  - → Objects are the elements of the partial order
  - → Morphisms represent the ≤ relation.
  - → Composition works because of the transitivity of ≤

E.g.

The partial order n, formed from the first n natural numbers

Here, n = 4



Outline

(1) An introduction to categories

- → Definitions
- → Some simple examples

(2) Motivations

- → Why is category theory so useful in mathematics?
- → Why is category theory relevant to software engineering?

you are here

(3) Enough category theory to get by

- → some important universal mapping properties
- → constructiveness and completeness

(4) Applying category theory to specifications

- → Specification morphisms
- → Modular Specifications
- → Tools based on category theory

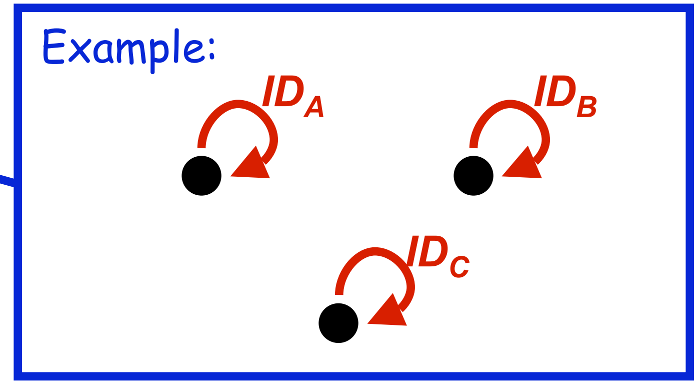
So what? *(for the mathematician)*

- **Category theory is a convenient new language**
  - → It puts existing mathematical results into perspective
  - → It gives an appreciation of the unity of modern mathematics
- **Reasons to study it**
  - → As a language, it offers economy of thought and expression
  - → It reveals common ideas in (ostensibly) unrelated areas of mathematics
  - → A single result proved in category theory generates many results in different areas of mathematics
  - → Duality: for every categorical construct, there is a dual, formed by reversing all the morphisms.
  - → Difficult problems in some areas of mathematics can be translated into (easier) problems in other areas (e.g. by using functors, which map from one category to another)
  - → Makes precise some notions that were previously vague, e.g. 'universality', 'naturality'

"To each species of mathematical structure, there corresponds a category, whose objects have that structure, and whose morphisms preserve it" - Goguen

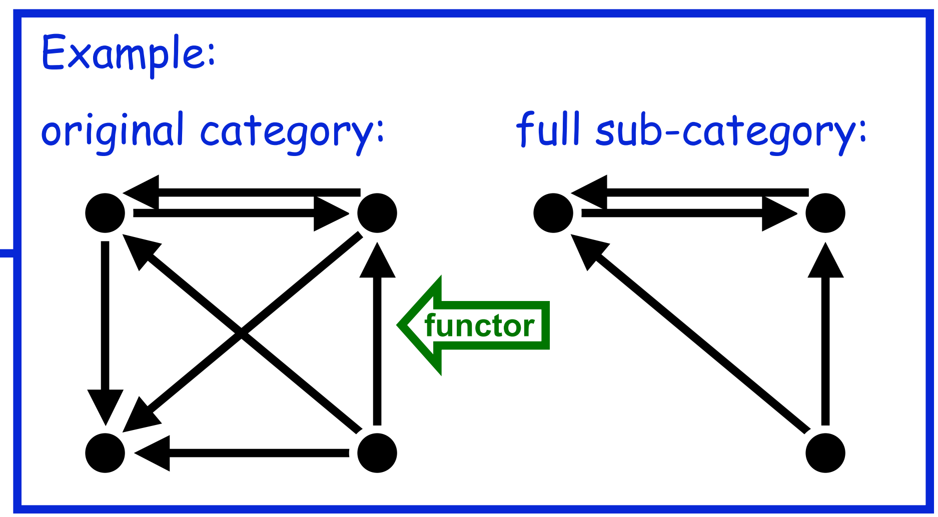
Some more definitions

- **Discrete category:**
  - → All the morphisms are identities



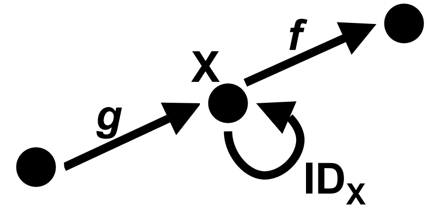
- **Connected category:**
  - → For every pair of objects, there is at least one morphism between them

- **Full sub-category:**
  - → A selection of objects from a category, together with *all* the morphisms between them.



Inverses and Isomorphisms

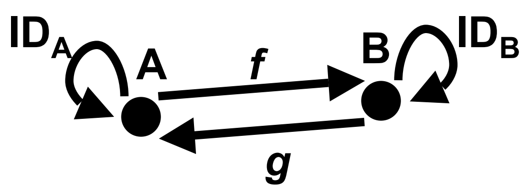
- **Identity morphism:**
  - → For each object X, there is an *identity morphism*, ID_X, such that:
  - → if f is a morphism with domain X, $f \circ ID_X = f$
  - → if g is a morphism with codomain X, $ID_X \circ g = g$



- **Inverse**
  - → g:B→A is an *inverse* for f:A→B if:

    $f \circ g = ID_B$

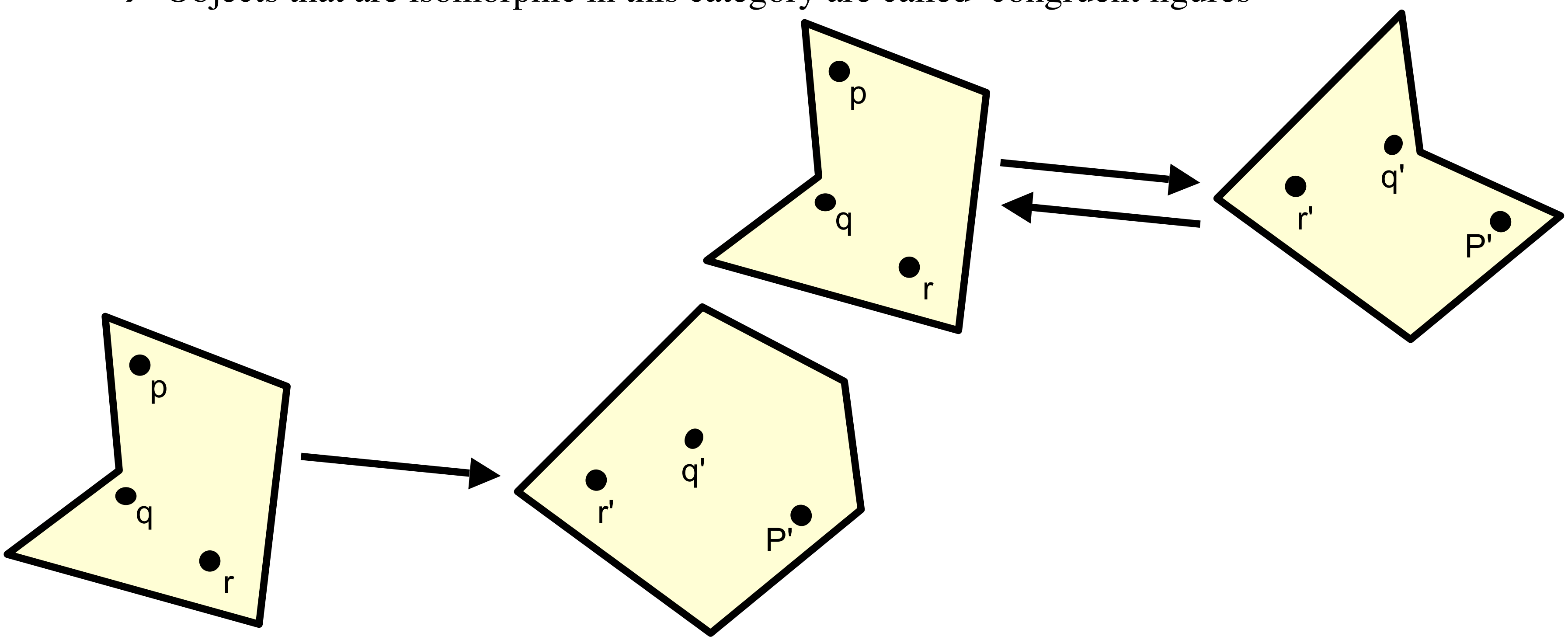
    $g \circ f = ID_A$
  - → If it exists, the inverse of f is denoted f^{-1}
  - → A morphism can have at most one inverse



- **Isomorphism**
  - → If f has an inverse, then it is said to be an *isomorphism*
  - → If f:A→B is an isomorphism, then A and B are said to be *isomorphic*

Example category 3

- **Category of geometric shapes (Euclid's category)**
  - → objects are polygonal figures drawn on a plane
  - → morphisms are geometric translations of all the points on the polygon such that distances between points are preserved.
  - → Objects that are isomorphic in this category are called 'congruent figures'



Example category 4

- **Category of algebras**
  - → Each object is a sort, with a binary function over that sort
  - → Each morphism is a translation from one algebra to another, preserving the structure

E.g.

$$(\Re,+) \xrightarrow{\text{exponentiation}} (\Re_{>0},\times)$$

$$(\Re,+) \xrightarrow{\text{doubling}} (\Re,+)$$

Works because $e^{(a+b)} = e^a \times e^b$

Works because $2(a+b) = 2a + 2b$

E.g.

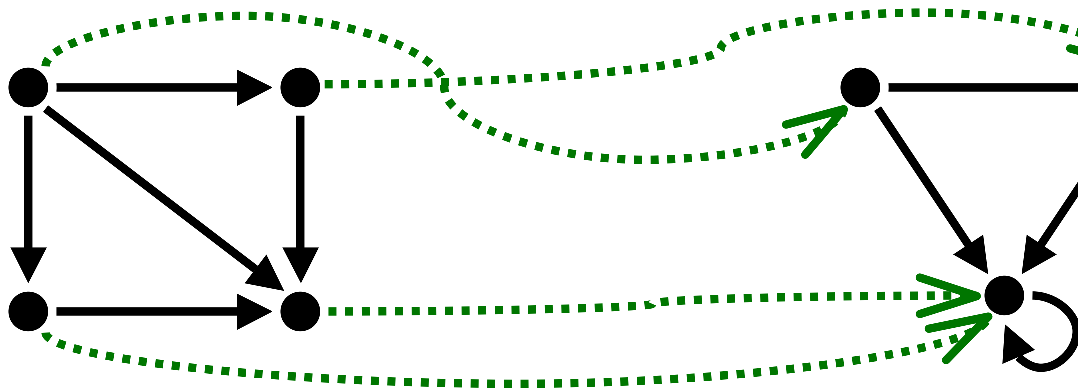
$$\begin{tikzcd} (\{\text{odd}, \text{even}\},+) \arrow[d] \\ (\{\text{pos}, \text{neg}\},\times) \end{tikzcd}$$

Functors

- **Definition of functor:**
  - → Consider the category in which the objects are categories and the morphisms are mappings between categories. The morphisms in such a category are known as *functors*.
  - → Given two categories, C and D, a functor F:C→D maps each morphism of C onto a morphism of D, such that:

    F preserves identities - i.e. if x is a C-identity, then F(x) is a D-identity

    F preserves composition - i.e $F(f \circ g) = F(f) \circ F(g)$



- **Example functor**
  - → *From* the category of topological spaces and continuous maps *to* the category of sets of points and functions between them

So what? *(for the software engineer)*

- **Category theory is ideal for:**
  - → Reasoning about structure and the mappings that preserve structure
  - → Abstracting away from details.
  - → Automation (constructive methods exists for many useful categorical structures)

- **Applications of Category theory in software engineering**
  - → The category of algebraic specifications - category theory can be used to represent composition and refinement
  - → The category of temporal logic specifications - category theory can be used to build modular specifications and decompose system properties across them
  - → Automata theory - category theory offers a new way of comparing automata
  - → Logic as a category - can represent a logical system as a category, and construct proofs using universal constructs in category theory (*"diagram chasing"*).
  - → The category of logics - theorem provers in different logic systems can be hooked together through 'institution morphisms'
  - → Functional Programming - type theory, programming language semantics, etc

Modularity in Software Engineering

- **Reasons for wanting modularization**
  - → Splitting the workload into workpieces
    "decompose the process"
  - → Splitting the system into system pieces (components)
    "decompose the implementation"
  - → Splitting the problem domain into separate concerns
    "decompose the requirements"

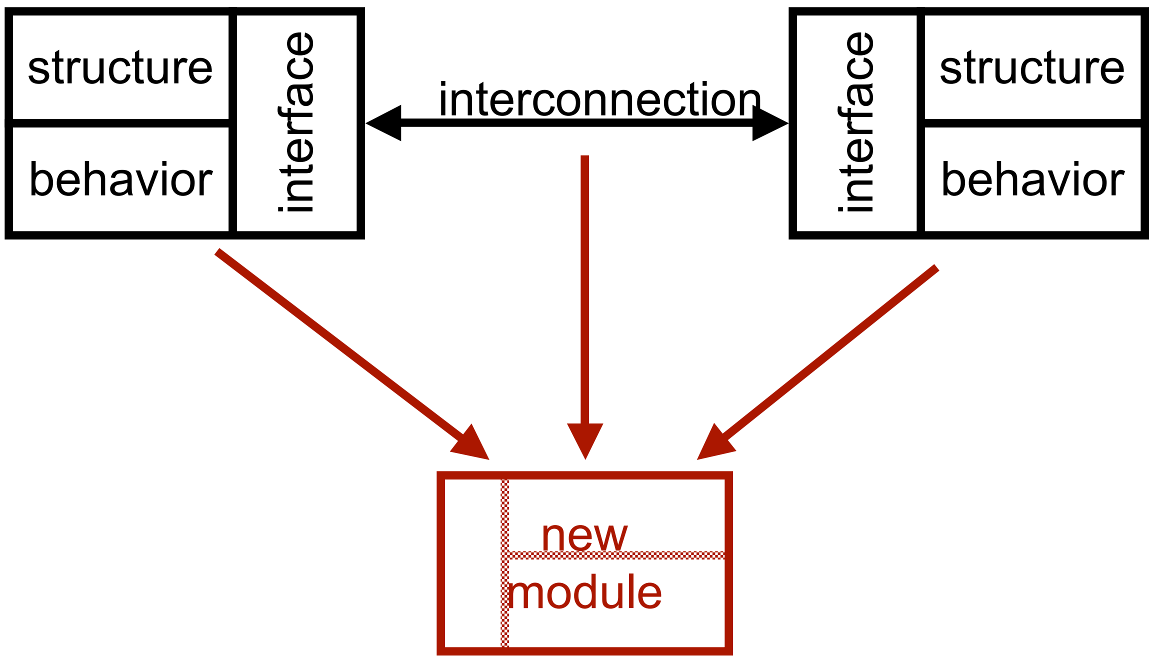
Most of the category theory work has addressed this one

- **Resulting benefits**
  - → Information hiding
  - → Compositional verification
  - → Compositional refinement

- **Generalizable approaches:**
  - → Semi-formal - *Viewpoints framework*
  - → Formal - *Category Theory*

Building blocks

- **Need to express:**
  - → *Modules* (Interface + Structure + Behavior)
  - → *Module Interconnections*
  - → *Operations on modules* (e.g. compose two modules to form a third)



Outline

(1) An introduction to categories

- → Definitions
- → Some simple examples

(2) Motivations

- → Why is category theory so useful in mathematics?
- → Why is category theory relevant to software engineering?

(3) Enough category theory to get by

- → some important universal mapping properties
- → constructiveness and completeness

you are here

(4) Applying category theory to specifications

- → Specification morphisms
- → Modular Specifications
- → Tools based on category theory

Enough Category Theory to get by...

- **Universal Constructs**
  - → General properties that apply to all objects in a category
  - → Each construct has a *dual*, formed by reversing the morphisms
  - → Examples:
    - – initial and terminal objects
    - – pushouts and pullbacks
    - – colimits and limits
    - – co-completeness and completeness

These are the building blocks for manipulating specification structures

- **Higher order constructs**
  - → Can form a category of categories. The morphisms in this category are called *functors*.
  - → Can form a category of functors. The morphisms in this category are called *natural transformations*.
  - → Can consider inverses of functors (and hence isomorphic categories). Usually, a weaker notion than isomorphism is used, namely *adjoint functors*.

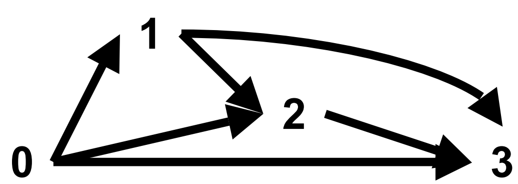
Initial and Terminal Objects

- **Initial objects**

  An object S is said to be ***initial*** if for every other object X there is exactly one morphism f:S→ X

- **Examples**

  → The number 0 in this category:



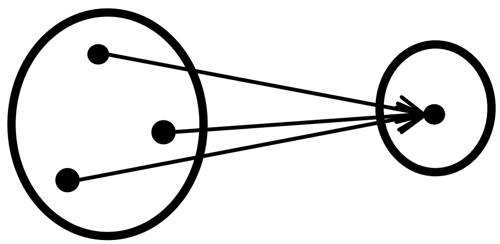
  → The empty set {} in the category of sets

- **Terminal objects**

  An object T is said to be ***terminal*** if for every other object X there is exactly one morphism f:X→T

- **Example**

  → Any singleton set in the category of sets

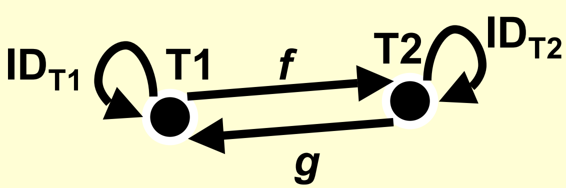


Uniqueness (up to isomorphism):

→ If T1 and T2 are both terminal objects, then there is exactly one morphism between them, and it is an isomorphism

→ Why? Because there is exactly one morphism each of f:T1→T2, g:T2→T1, h:T1→T1, and j:T2→T2, where h and j are identities.

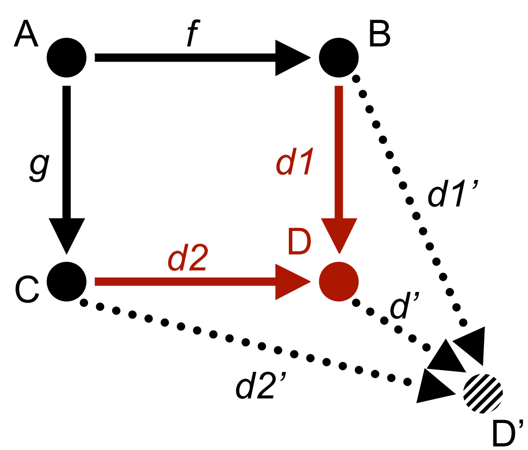
→ Same applies to initial objects



Pushouts and Pullbacks

- **Pushout**

  The *pushout* for two morphisms f:A→B and g:A→C is an object D, and two morphisms d1:B→D and d2:C→D, such that the square commutes…

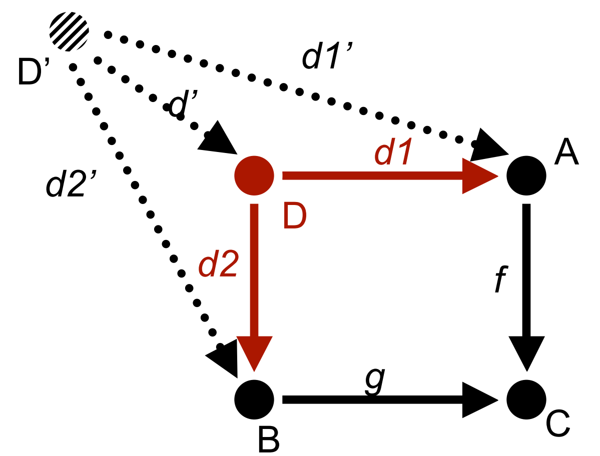


  … and D is the initial object in the full subcategory of all such candidates D'

  (I.e. for all objects D' with morphisms d1' and d2', there is a unique morphism from D to D')

- **Pullback**

  The *pullback* for two morphisms f:A→C and g:B→C is an object D, and two morphisms d1:D→A and d2:D→B, such that the square commutes…

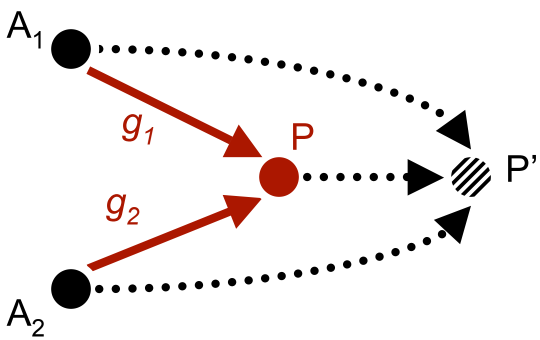


  … and D is the terminal object in the full subcategory of all such candidates D'

Products and Coproducts

- **Coproduct**

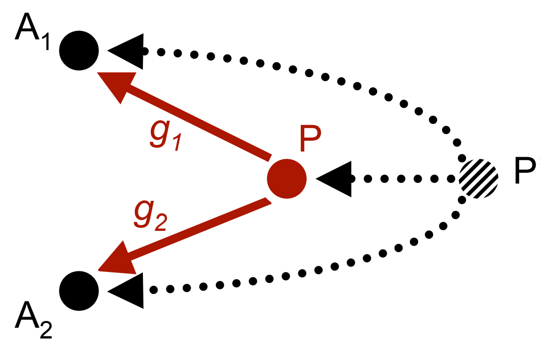
  The coproduct of a family of objects A_i is an object P and a set of morphisms $g_i:A_i \rightarrow P$



  … and P is the initial object in the full subcategory of all such candidates P'

- **Product**

  The product of a family of objects A_i is an object P and a set of morphisms $g_i:P \rightarrow A_i$



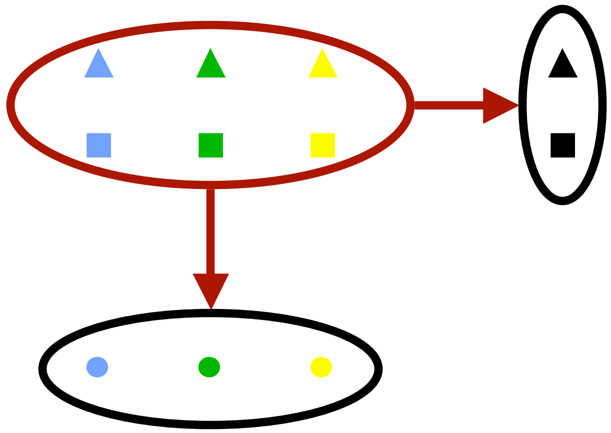
  … and P is the terminal object in the full subcategory of all such candidates P'

- **Coproduct vs. Pushout**
  - → Pushout is a universal property of any two morphisms with a common domain
  - → Coproduct is a universal property of any set of objects

- **Product vs. Pullback**
  - → Pullback is a universal property of any two morphisms with a common codomain
  - → Product is a universal property of any set of objects

Example products

- **In the category of sets:**
  - → constructed as the cartesian product

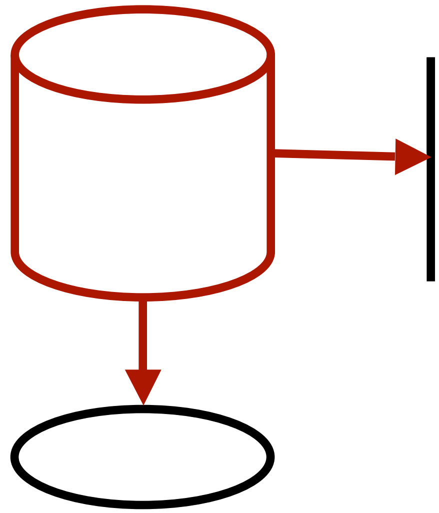


- **In the category of logical propositions:**

$$\begin{tikzcd} a\wedge b \arrow[r] \arrow[d] & b \\ a & \end{tikzcd}$$

$$\begin{tikzcd} p' \arrow[dr, dotted] \arrow[drr, dotted] \arrow[ddr, dotted] & & \\ & a\wedge b \arrow[r] \arrow[d] & b \\ & a & \end{tikzcd}$$

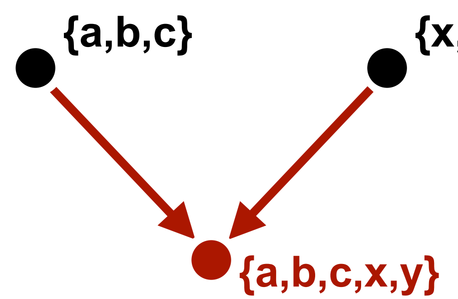
- **In the category of geometric spaces:**



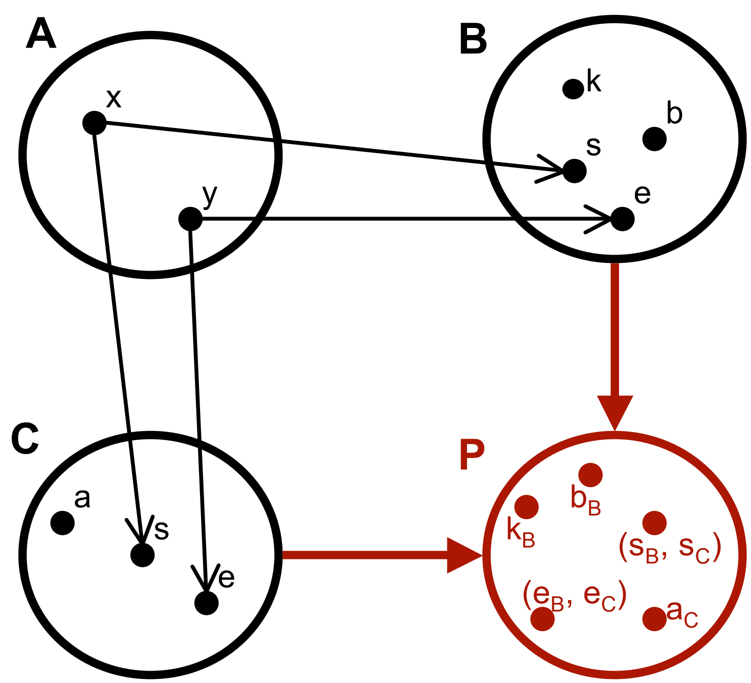
In any given category, some products might not exist. It is useful to know whether they all do.

Example co-product & pushout

- **Coproducts on the category of sets:**
  - → Constructed by taking the disjoint sum



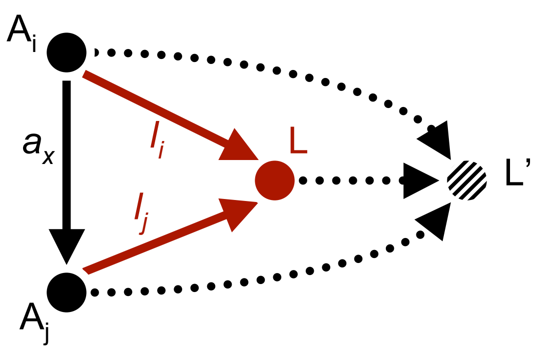
- **Pushouts on the category of sets:**
  - → Union of:
  - → Pairs of elements from B and C that are the images of the same element in A
  - → Plus all the remaining elements of B and C



Limits and Colimits

- **Colimits**
  - → initial objects, pushouts and coproducts are all special cases of colimits.
  - → Colimits are defined over any diagram

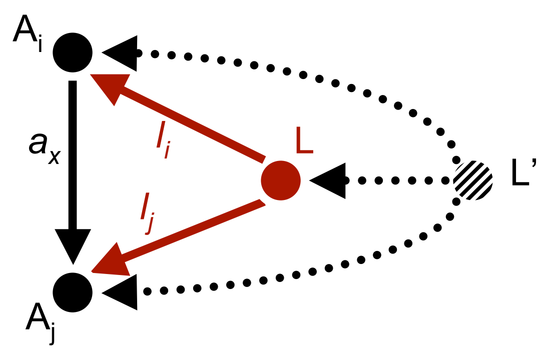
For any diagram containing objects A_i and morphisms a_i, the *colimit* of this diagram is an object L and a family of morphisms l_i, such that for each l_i: $A_i \rightarrow L$, l_j: $A_j \rightarrow L$, and $a_x : A_i \rightarrow A_j$, then $l_j \circ a_x = l_i$



… and L is the initial object in the full subcategory of all such candidates L'

- **Limits**
  - → terminal objects, pullbacks and products are all special cases of limits.
  - → Limits are defined over any diagram

For any diagram containing objects A_i and morphisms a_i, the *limit* of this diagram is an object L and a family of morphisms l_i, such that for each $l_i : L \rightarrow A_i$, $l_j : L \rightarrow A_j$, and $a_x : A_i \rightarrow A_j$, then $a_x \circ l_i = l_j$



… and L is the terminal object in the full subcategory of all such candidates L'

Completeness and Co-completeness

- **It is useful to know for a given category which universal constructs exist:**
  - → If a category has a terminal object and all pullbacks exist, then all finite limits exist
    - – Hence it is finitely complete
  - → If a category has an initial object and all pushouts exist, then all finite colimits exist
    - – Hence it is finitely cocomplete

- **Proofs are usually constructive**
  - → I.e. give a method for computing all pullbacks (pushouts)
  - → The constructive proof is the basis for automated generation of limits (colimits)

- **Obvious application**
  - → If your objects are specifications, then:
    - – colimits are the integration of specifications
    - – limits are the overlaps between specifications

Outline

(1) An introduction to categories

- → Definitions
- → Some simple examples

(2) Motivations

- → Why is category theory so useful in mathematics?
- → Why is category theory relevant to software engineering?

(3) Enough category theory to get by

- → some important universal mapping properties
- → constructiveness and completeness

(4) Applying category theory to specifications

- → Specification morphisms
- → Modular Specifications
- → Tools based on category theory

you are here

(Recall...) Algebraic Specifications

- **A signature is a pair <S, Ω>**

  where S is a set of sorts, and Ω is a set of operations over those sorts
- **A specification is a pair <Σ, Φ>**

  describes algebras over the signature Σ that satisfy the axioms Φ

Signature

Body

```
Spec Container
    sort Elem, Cont
    op empty: Cont
    op single: Elem -> Cont
    op merge: Cont, Cont -> Cont
    axiom merge(empty, e) = e
    axiom merge(e, empty) = e
end-spec
```

- **Semantically:**
  - → We are modeling programs as algebras
  - → A specification defines a class of algebras (programs)

Specification morphisms

- **Specfication morphisms**
  - → Consider the category in which the objects are specifications
  - → The morphisms translate the vocabulary of one specification into the vocabulary of another, preserving the (truth of the) axioms
- **Actually, there are two parts:**
  - → *Signature morphism:* a vocabulary mapping
    - – maps the sorts and operations from one spec to another
    - – must preserve the rank of each operation
  - → *Specification morphism:* a signature morphism for which each axiom of the first specification maps to a theorem of the second specification
- **Proof obligations**
  - → There will be a bunch of proof obligations with each morphism, because of the need to check the axioms have been translated into theorems
  - → A theorem prover comes in handy here.

Example

```
Spec Container
    sort Elem, Cont
    op empty: Cont
    op single: Elem -> Cont
    op merge: Cont, Cont -> Cont
    axiom merge(empty, e) = e
    axiom merge(e, empty) = e
end-spec
```

These comprise the signature morphism (Note each spec has it's own namespace)

```
Spec List
    sort Elem, List
    op null: List
    op single: Elem -> List
    op append: List, List -> List
    op head: List -> Elem
    op tail: List -> List
    axiom head(single(e)) = e
    axiom tail(single(e)) = null
    axiom append(single(head(l)), tail(l)) = l
end-spec
```

These axioms must be true down here (after translation)

What do we gain?

- **Three simple *horizontal composition* primitives:**
  - → *Translate:* an isomorphic copy (just a renaming)
    - – can test whether two specifications are equivalent
  - → *Import:* include one specification in another (with renaming)
    - – for extending specifications with additional services
  - → *Union (colimit):* Compose two specifications to make a larger one
    - – system integration

- **One simple *vertical composition* primitive:**
  - → *refinement:* mapping between a specification and its implementation
    - – introduce detail, make design choices, add constraints, etc.
    - – (may want to use different languages, e.g. refinement is a program)

Example colimit (pushout)

```
Spec Container
    sort A, B
    op x: B
end-spec
```

```
Spec List
    sort Elem, List
    op null: List
    op head: List -> Elem
    op tail: List -> List
    op cons: Elem, List -> List
    axiom head(cons(e, l)) = e
    axiom tail(cons(e, l)) = l
    axiom cons(head(l), tail(l)) = l
    axiom tail(cons(e, null)) = null
end-spec
```

```
Spec Container
    sort Elem, Cont
    op empty: Cont
    op single: Elem -> Cont
    op merge: Cont, Cont -> Cont
    axiom merge(empty, e) = e
    axiom merge(e, empty) = e
end-spec
```

New spec is lists with two new operations, "single" and "merge"

(Recall…) Temporal Logic Specs

- **A signature is a pair <S, Ω>**

  where S is a set of sorts, and Ω is a set of operations over those sorts

- **A specification is a 4-tuple <Σ, ATT, EV, AX>**

  Σ is the signature

  ATT is a set of attributes

  EV is a set of events

  AX is a set of axioms expressed in temporal logic

These three comprise the vocabulary of the specification

Assume some usual temporal logic operators, e.g.
- □ always
- ◇ eventually

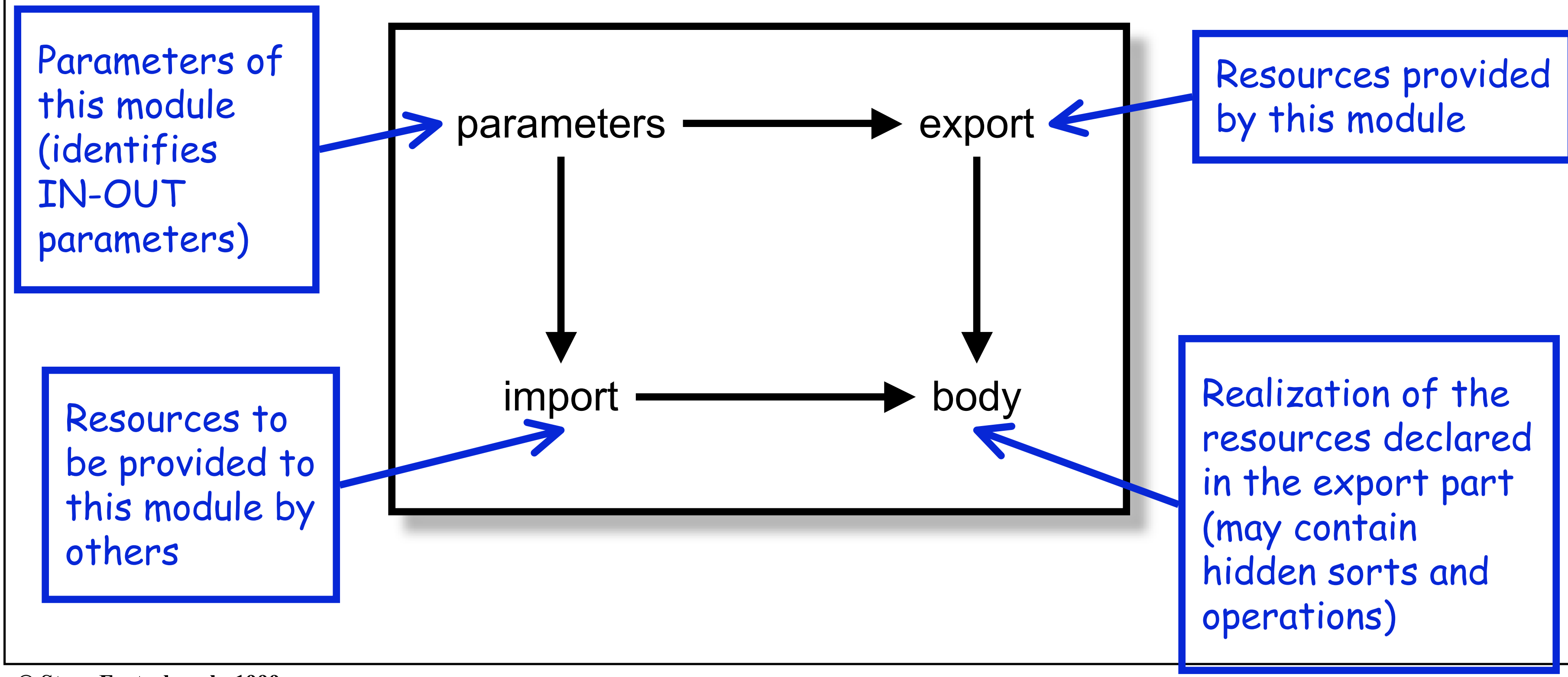
- **Semantically:**
  - → We are modeling programs as state machines
  - → A specification describes a class of state machines that obey the axioms

- **(A minor complication)**
  - → Need to worry about locality of events

Expressing modules

- **Want to generalize the notion of a module**
  - → Explicitly declare interfaces, with constraints on imported and exported resources
  - → Hence the interface itself is a specification (actually 2 specifications)

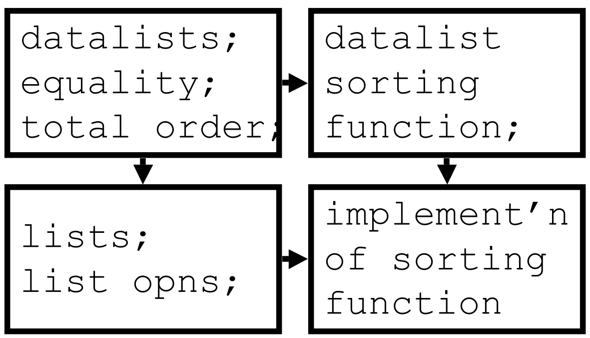
  (Ehrig & Mahr use algebraic specs; Michel & Wiels use temporal logic specs)



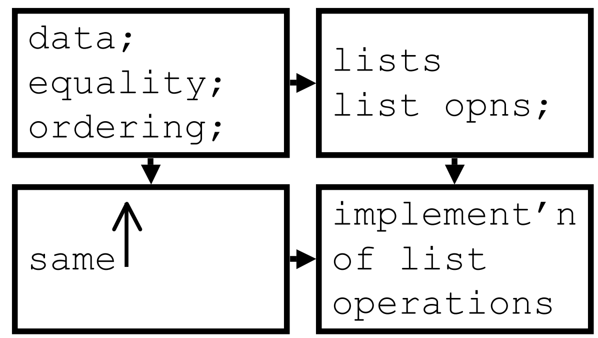
Examples

- **The approach works for many different kinds of module:**

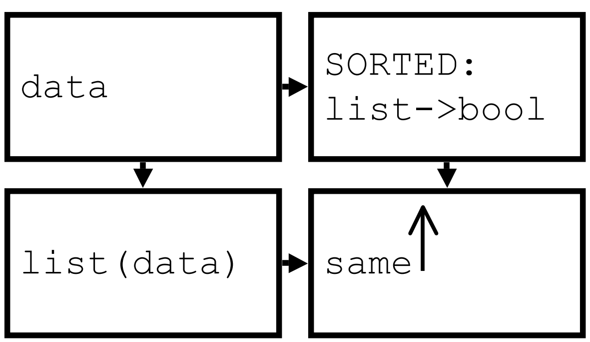
E.g. function modules



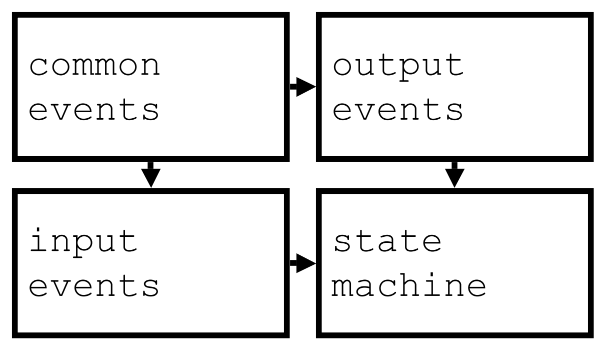
E.g. data types



E.g. predicates

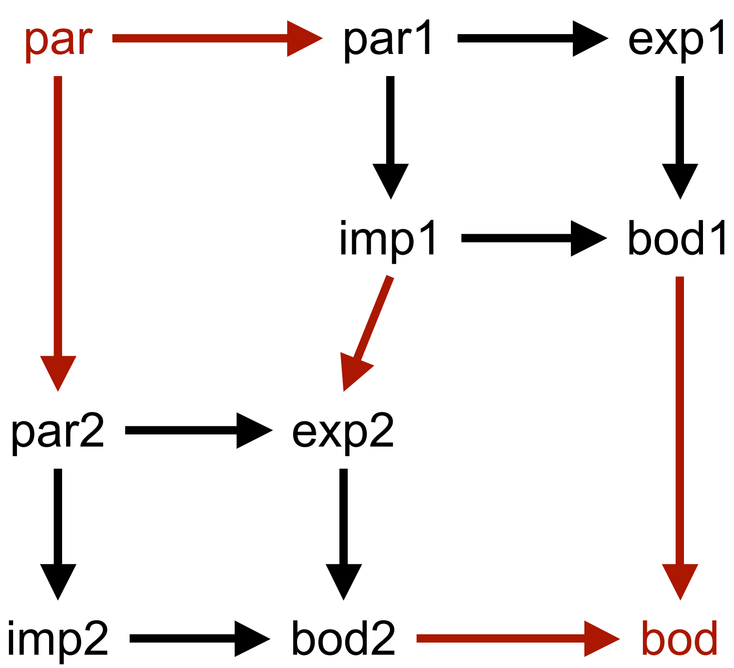


E.g. state machines

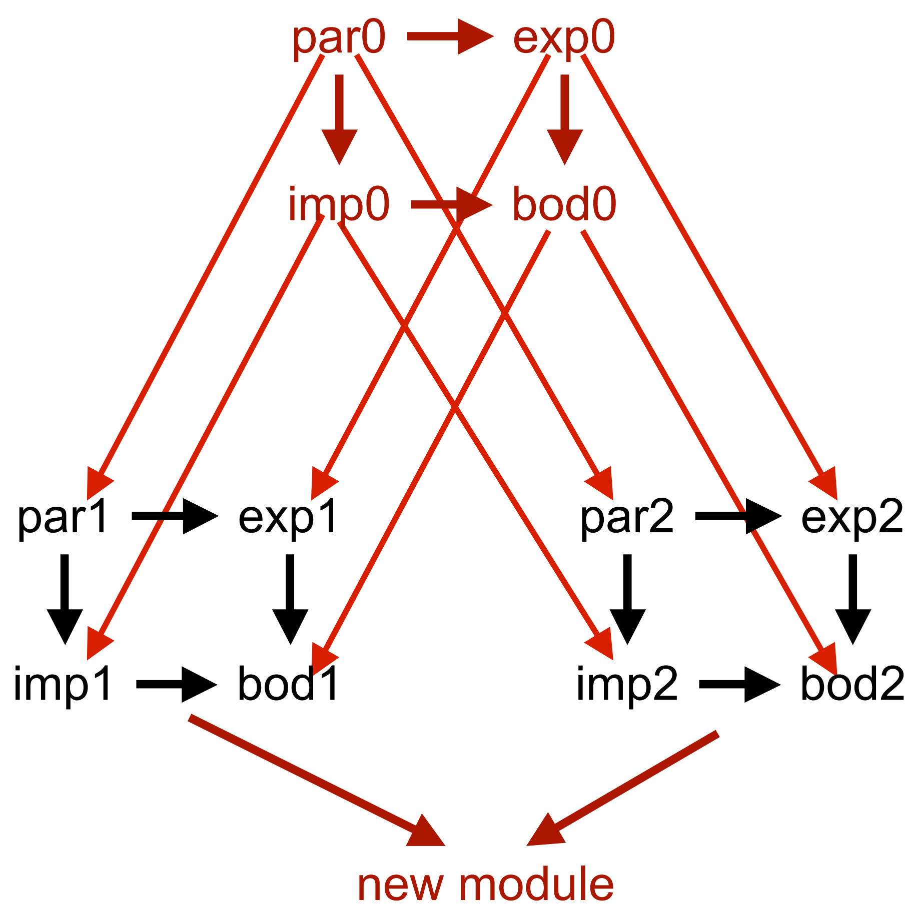


Composing modules

E.g. import ("uses"):



E.g. union (colimit):



Advanced Topics

- **Logic engineering**
  - → Language translation
    - – from one logic to another
    - – from one specification language to another
  - → Aim is to characterize logics as:
    - – signatures (alphabet of non-logical symbols)
    - – consequence relations
  - → Then an *institution morphism* allows you to translate from one logic to another whilst preserving consequence

- **Natural Transformations of refinements**
  - → If a system specification is a category, and the relationship between the specification and its refinement is a functor…
  - → …then the relationship between alternative refinements of the same specification is a natural transformation.

(suggested) Future Research Issues

- **Compositional Verification in Practice**
  - → E.g. How much does the choice of modularization affect it
  - → Which kinds of verification properties can be decomposed, and which cannot?
  - → How do we deal systemic properties (e.g. fairness)
- **Evolving Specifications**
  - → How do you represent and reason about (non-correctness preserving) change?
  - → How resilient is a modular specification to different kinds of change request
- **Dealing with inconsistencies**
  - → Specification morphisms only work if the specifications are consistent
  - → Can we weaken the "correct by construction" approach?

Summary

- **Category Theory basis**
  - → Simple definition: class of objects + class of arrows (morphisms)
  - → A category must obey identity, composition and associativity rules
- **Category theory is useful in mathematics…**
  - → Unifying language for talking about many different mathematical structures
  - → Provides precise definition for many abstract concepts (e.g. isomorphism)
  - → Framework for comparing mathematical structures
- **Category theory is useful in software engineering**
  - → Modeling and reasoning about structure
  - → Provides precise notions of modularity and composition
  - → Specification morphisms relate vocabulary and properties of specifications
  - → Constructive approach lends itself to automation

Answer to challenge question:

YES!

(proof left as an exercise for the audience)*