Catecon
The Categorical Console

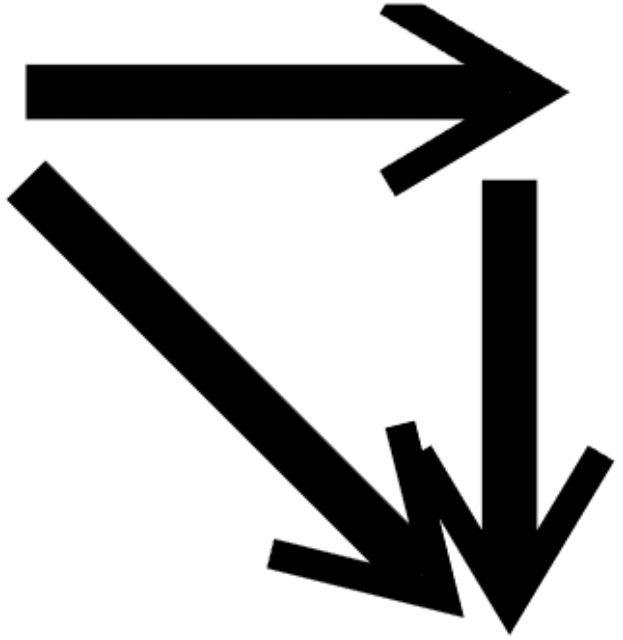
What is it?

What does it do?

And how does it do it?

Harry Dole

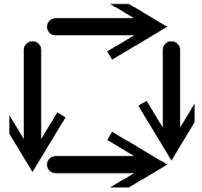
June 23, 2002



What Is Catecon?

Online sharing of categorical diagrams and programming

What is it?

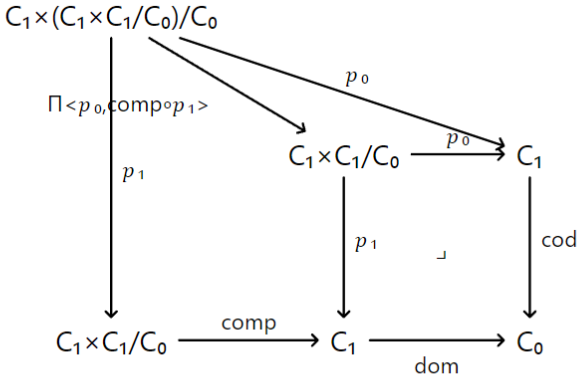


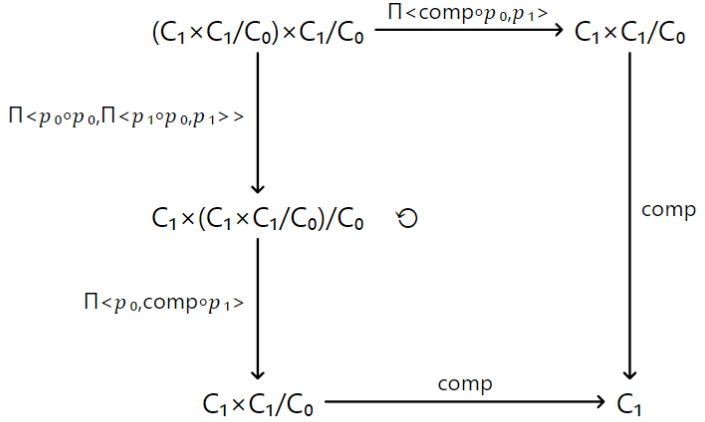
- https://catecon.net
- Use morphisms to draw categorical diagrams in a web browser
  - Reference other user's diagrams
  - Share your diagrams on the web
- Describe mathematical structures diagrammatically using:
  - Categorical operators: composition, product, coproduct, hom, lambda, pullbacks, …
  - Assertions, quantifiers
- Use definitions and prove theorems with commutative diagrams
- View morphism string graphs
- Use injections and projections as reference morphisms to easily construct complex morphisms
- Generate Javascript/C++ code from appropriate morphisms
- MIT license
- GitHub: https://github.com/hdole/catecon.git

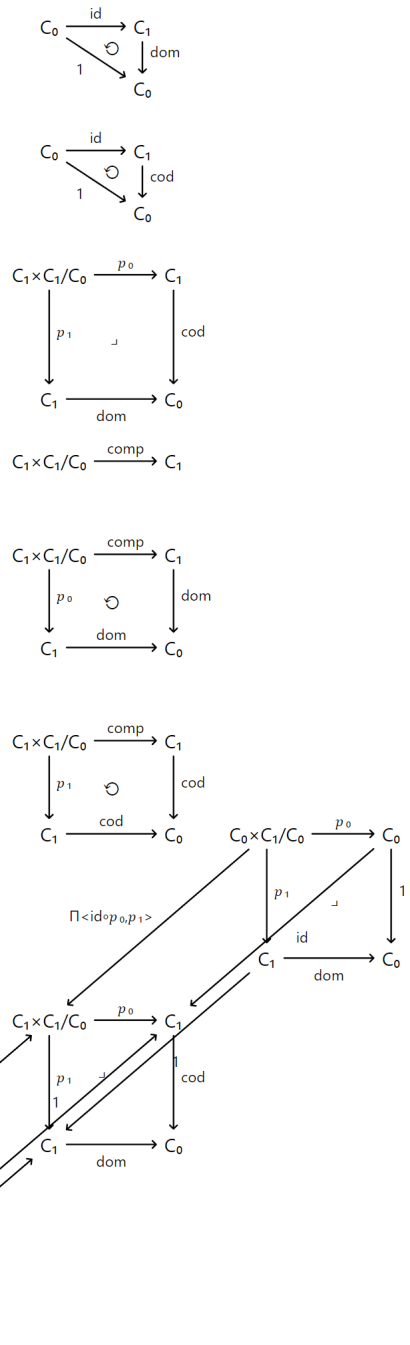
Example: Internal Definition of Category

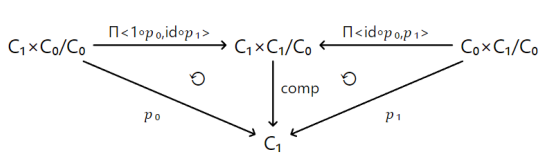
https://catecon.net/d/hdole/category

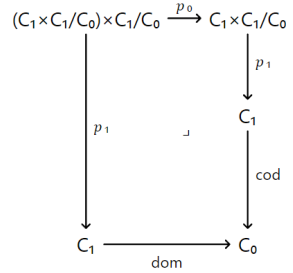
https://youtu.be/GlzivTGXC7I

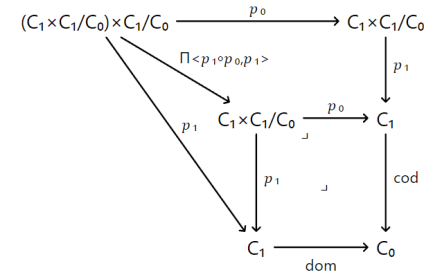


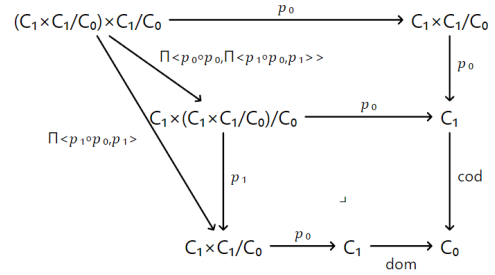


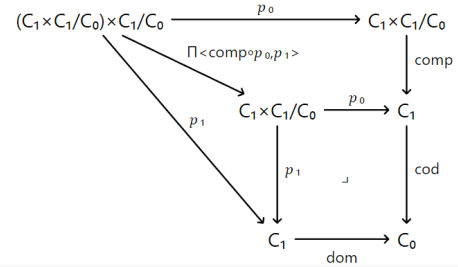






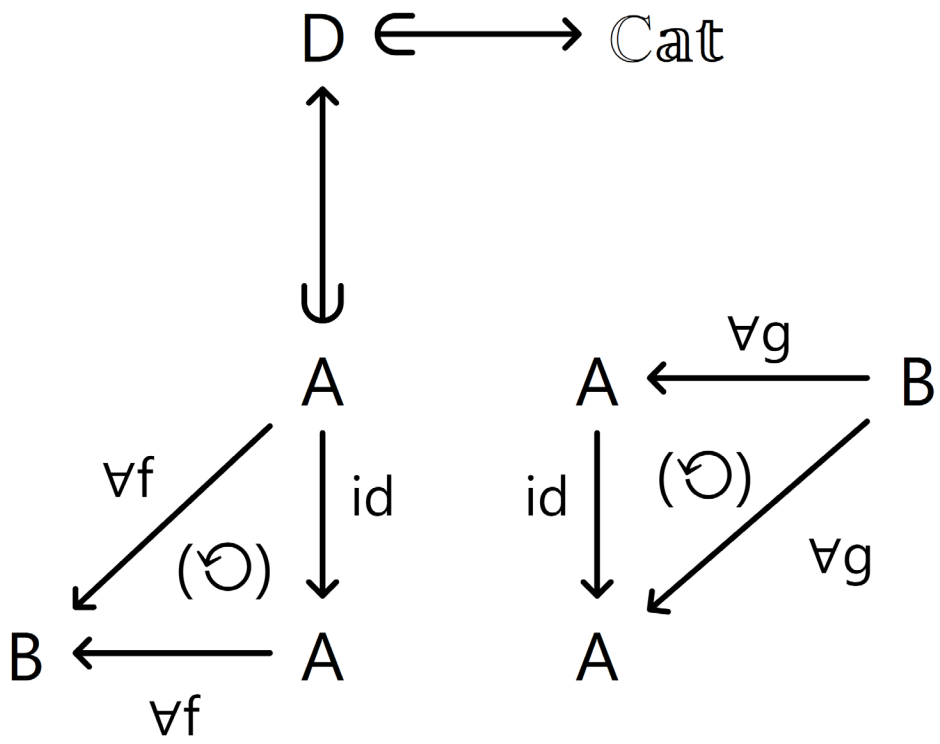


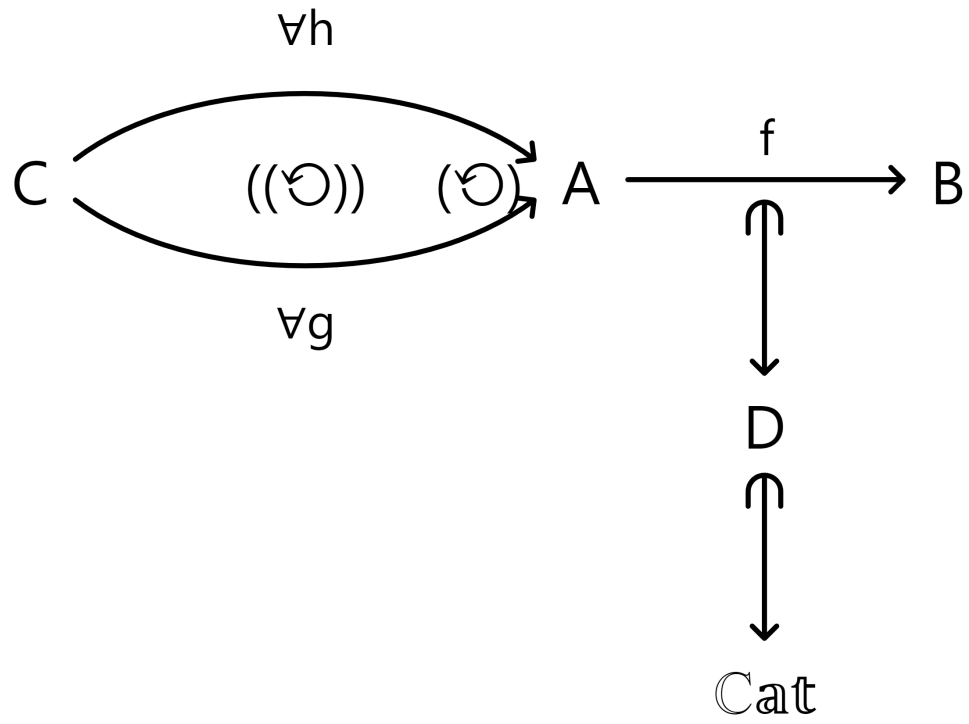




Example: Definitions of Identity and Monomorphism

- Parentheses denote depth in expression



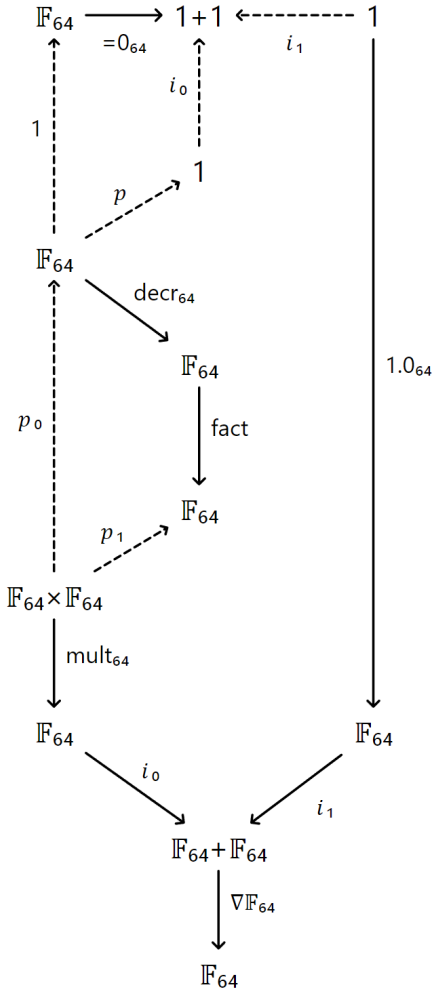


https://catecon.net/d/hdole/identity

https://catecon.net/d/hdole/monomorphism

Example: Generate C++ Code

- Projections as references assemble inputs
- Injections as references determine branching
- String graphs give variables



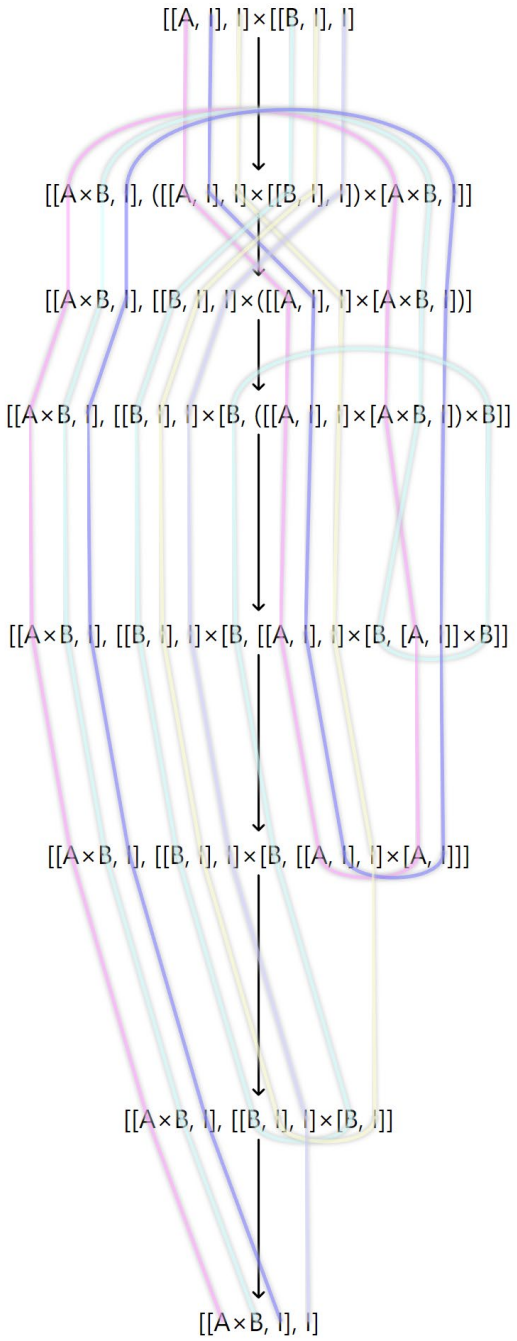

```
// hdole/floats/F64 --> hdole/floats/F64
//
void fun_3f82037f6f2be1de4fca3175ae74ac7179683323c2e015aea8d7164d5bbf3868(const hdole_floats_F64 var_2, hdole_floats_F64 & var_3)
{
    hdole_floats_CPo_Br__n_1_c__n_1_rB_oPC var_4;
    hdole_floats_F64 var_5;
    var_5 = 0.0;
    var_4 = var_2 == var_5;
    if (var_4)
    {
        var_3 = 1.0;
    }
    else
    {
        hdole_floats_F64 var_6;
        hdole_floats_F64 var_7;
        hdole_floats_F64 var_8;
        var_8 = 1.0;
        var_7 = var_2 - var_8;
        fun_3f82037f6f2be1de4fca3175ae74ac7179683323c2e015aea8d7164d5bbf3868(var_7, var_6);
        var_3 = var_2 * var_6;
    }
}
//
// Composite
// hdole/factorial/Cm{hdole/cpp/stdinToF64,fact,hdole/cpp/F64ToStdout}mC
// hdole/cpp/stdin --> hdole/cpp/stdout
//
void fun_37e66ac7b9e52326c0f779dbf081b0ea8a0465a7b161d01c69b0197a26532695()
{
    hdole_floats_F64 var_0;
    hdole_floats_F64 var_1;
    std::cin >> var_0;
    fun_3f82037f6f2be1de4fca3175ae74ac7179683323c2e015aea8d7164d5bbf3868(var_0, var_1);
    std::cout << var_1;
}
int main(int argc, char ** argv)
{
    try
    {
        if (argc == 2 && (strcmp("-h", argv[1]) || strcmp("--help", argv[1])))
        {
            std::cout << "Diagram: hdole/factorial" << std::endl;
            std::cout << "Morphism: hdole/factorial/Cm{hdole/cpp/stdinToF64,fact,hdole/cpp/F64ToStdout}mC" << std::endl;
            return 1;
        }
        fun_37e66ac7b9e52326c0f779dbf081b0ea8a0465a7b161d01c69b0197a26532695();
        return 0;
    }
    catch(std::exception x)
    {
        std::cerr << "An error occurred" << std::endl;
        return 1;
    }
}
```

https://catecon.net/d/hdole/factorial

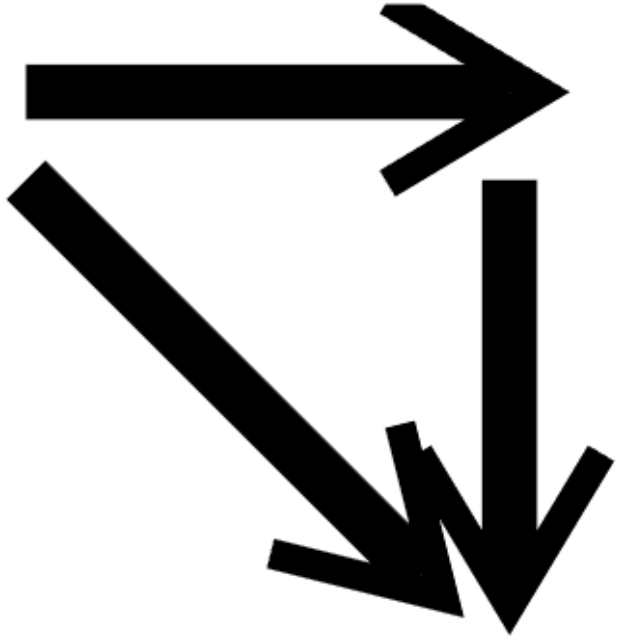
https://youtu.be/3E78dKBT5DU

Example: String Graphs

- Catecon keeps track of which term in an expression connects to which other ones
- Shown are the two Arens multiplications



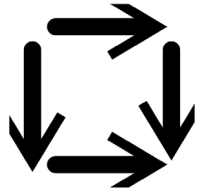
https://catecon.net/d/hdole/graph



What Does Catecon Do?

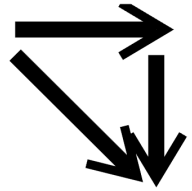
Categorically, that is

What Does Catecon Do

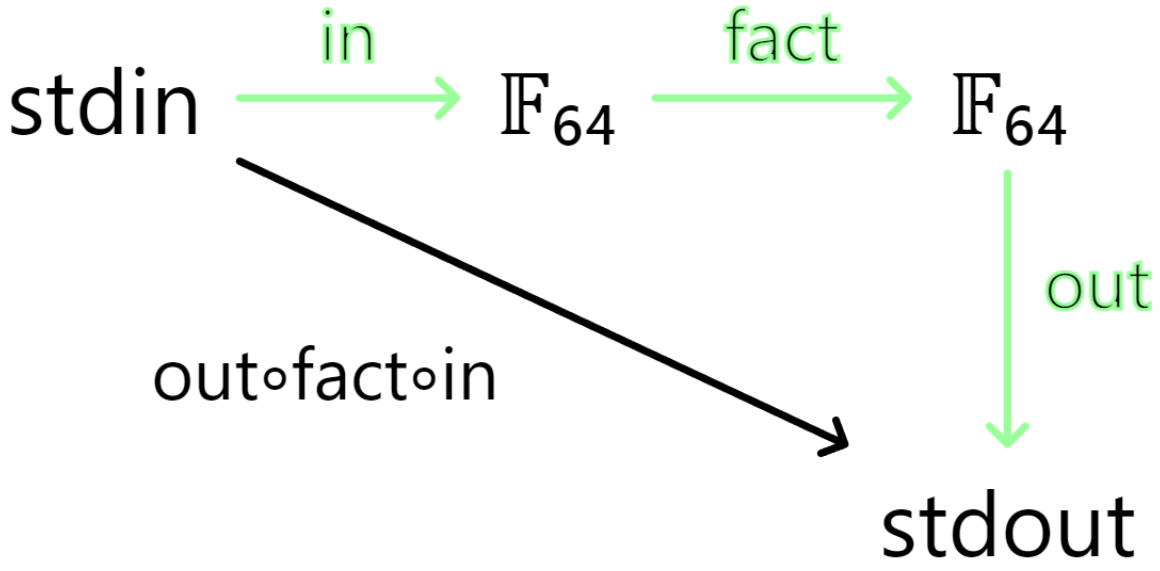


- Draw diagrams
- Actions supported:
  - Composition, product, coproduct, projections, injections, product/coproduct assemblies, lambda, distribute, hom, pullback (others TBD), plus recursion
- Naming of objects or morphisms
- String graphs
- Assign assertions to cells of diagrams to express commutativity
- Declare definitions and instantiate them in other categories
  - Attach quantifiers to morphisms and adjust depth of expression
- Declare theorems between definitions, create the corresponding diagrams from the target definition, and show commutativity
- Attach code to bare morphisms: Currently Javascript and C++
  - Construct complex morphisms and emit code

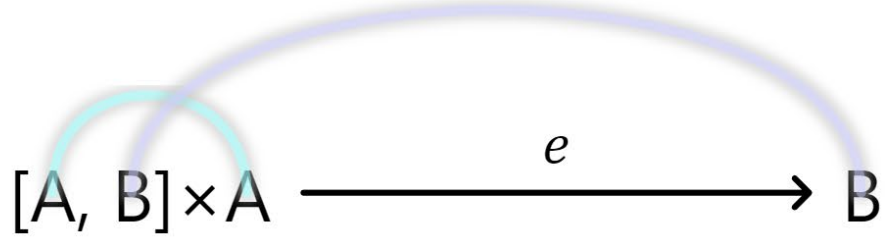
Composition

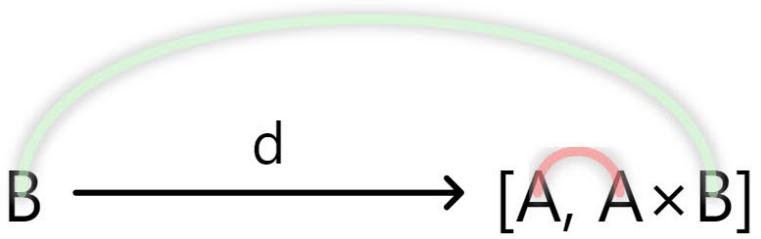


- If a user selects an appropriately connected sequence of morphisms, then the composite can be formed
- The composite morphism can then be reused in the diagram and other diagrams that reference this one

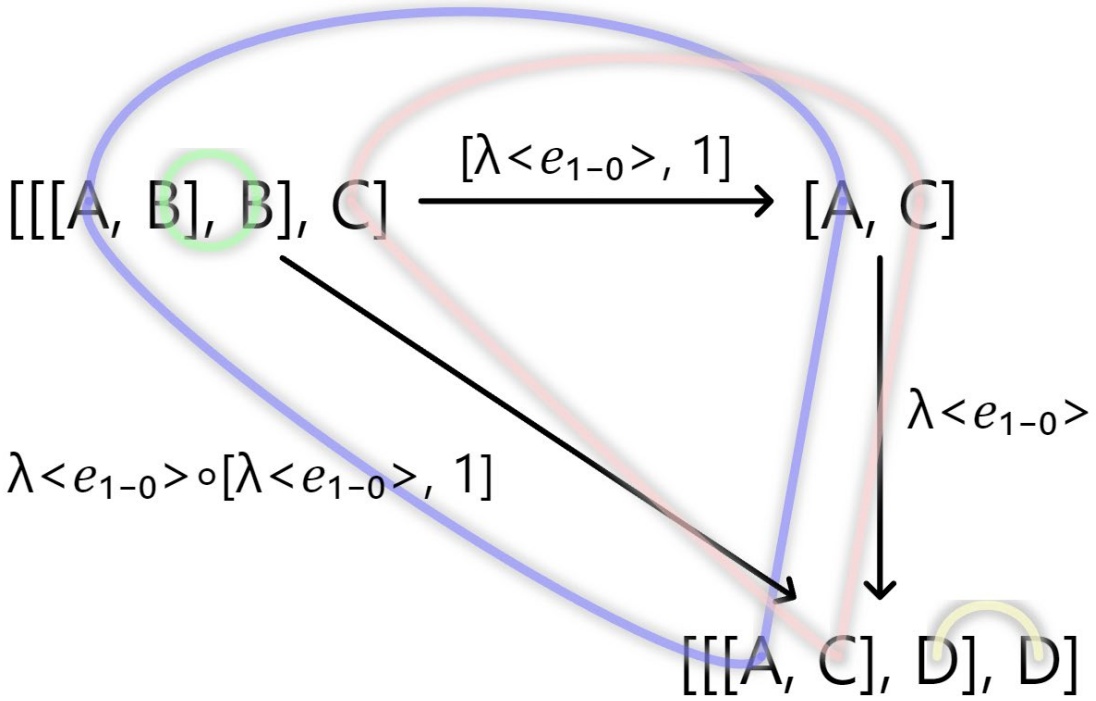


String Graphs





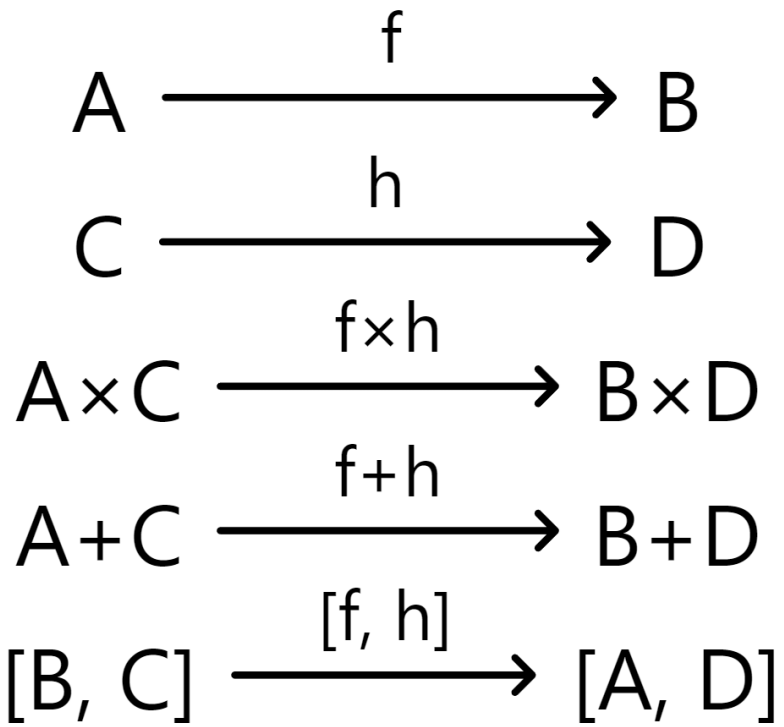
- Simple string graphs connecting terms between a morphism's domain and codomain can be shown



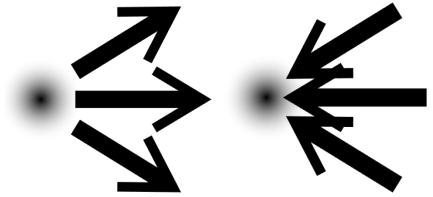
Product, Coproduct, Hom

$\times + [\_,\_]$

- Given a list of objects (or morphisms) in the same category, form the product or coproduct thereof
  - For hom, only two

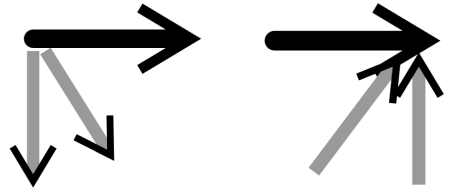


Factor Morphisms

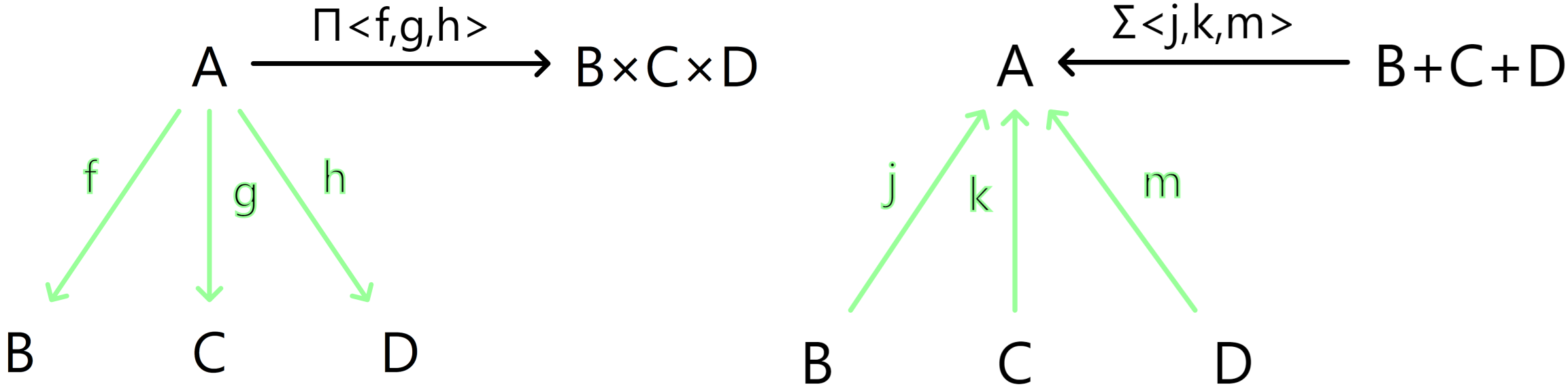


- Factor morphisms attempt to describe all those morphisms given an object and a list of its factors
  - Duality gives product vs coproduct
  - Currently associativity given by product assemblies and factor morphisms
- Covers: identities, projections, injections, twist, delta, fold
- Given the object A, the list A describes the identity on A
- Given A, the list A, A describes $A \to A \times A$ or $A + A \to A$ depending on duality
- Given $S \times T$, the list T, S describes the twist morphism $t: S \times T \to T \times S$
- Terminal, initial objects can be added as in $A \to A \times 1$ or $A \to A \times 0$

Product/Coproduct Assemblies



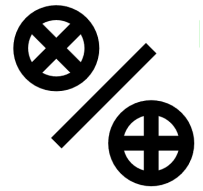
- Select a list of morphisms with common domain (or codomain) and assemble a morphism with codomain the product of the individual codomains (or dually)



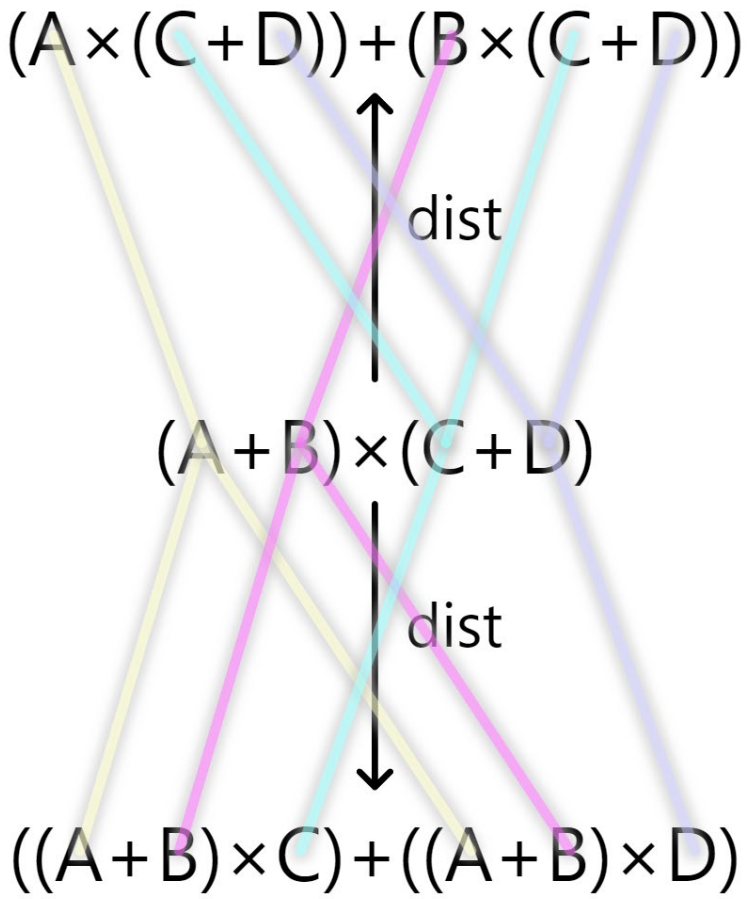
Lambda Morphisms

- Lambda morphisms describe morphisms derived from a single given morphism using currying
- For example
  - The identity [A, B]→[A, B] can be curried into the evaluation e:[A,B]×A→B
  - The identity A×B→A×B can be curried into the morphism d:A→[B,A×B]

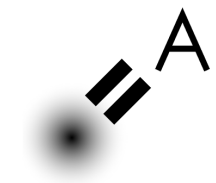
Distribute Morphisms



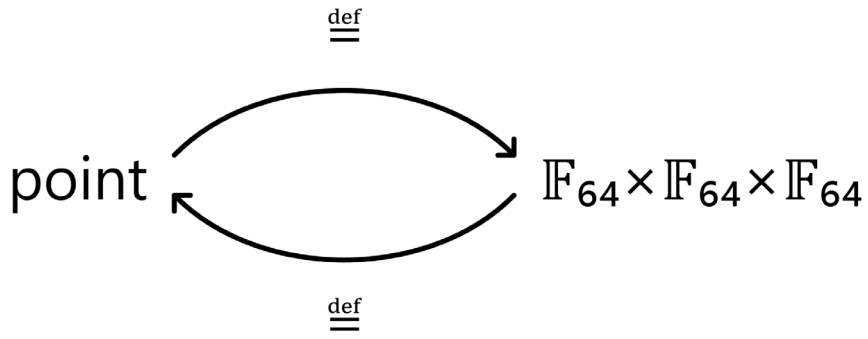
- Given an object, if it meets the pattern for left or right distribution of a product over a coproduct, then a distribution morphism may be created for left or right
- The string graphs are shown for the left and right distributions of (A+B)×(C+D)

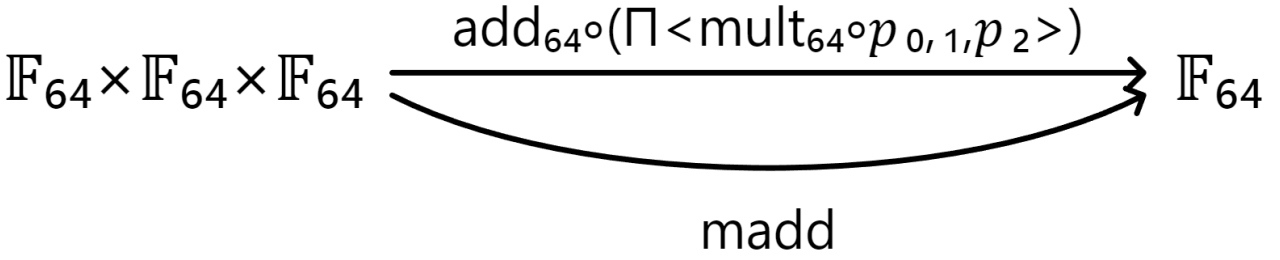


Named Objects and Morphisms



- Give complex objects or morphisms a simple name
- A named object creates two morphisms
  - For example, ‘point’ is an object with two morphisms connecting to the base term $\mathbb{F}_{64}\times\mathbb{F}_{64}\times\mathbb{F}_{64}$
- For a named morphism, a parallel morphism is created with a commutative cell
- Catecon sees them as same

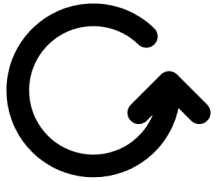




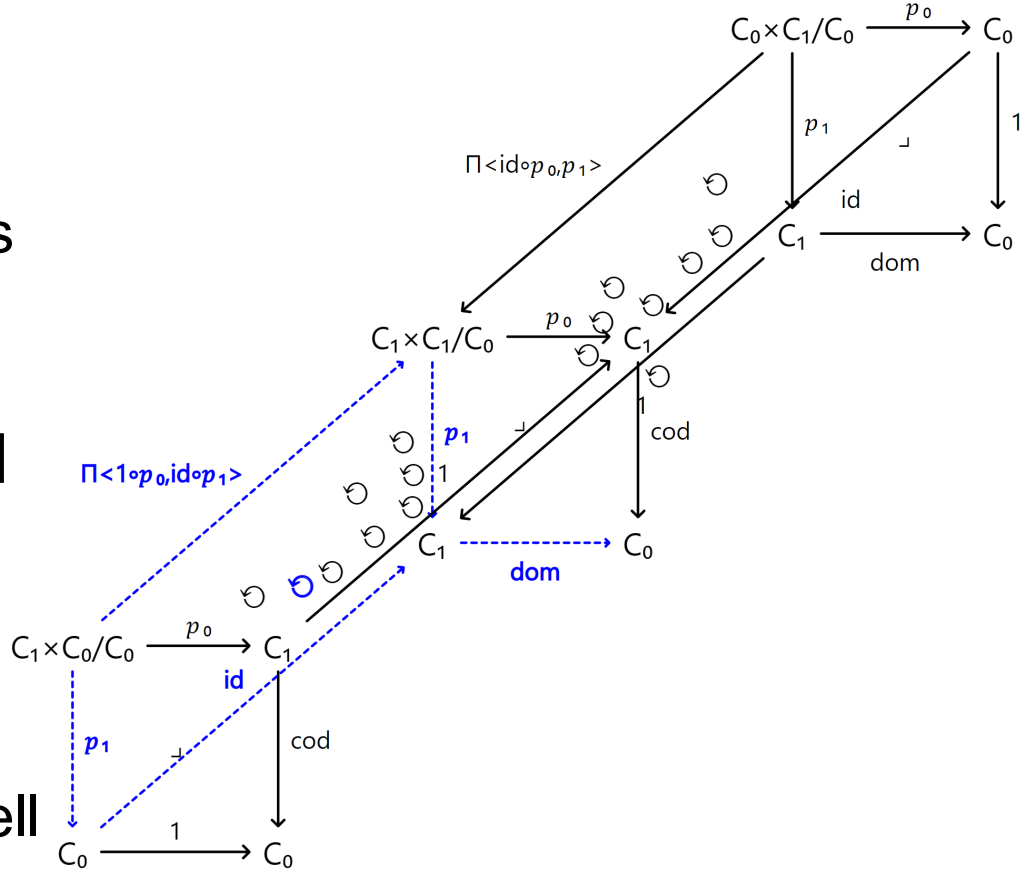
Recursion

- A bare morphism may have another morphism set as its recursor
  - Then no longer bare
- Recursor must have same target domain and codomain as selected morphism
- Recursor must use selected morphism somewhere in its expression

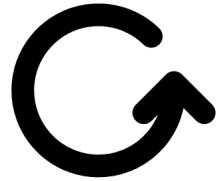
Blobs and Cells



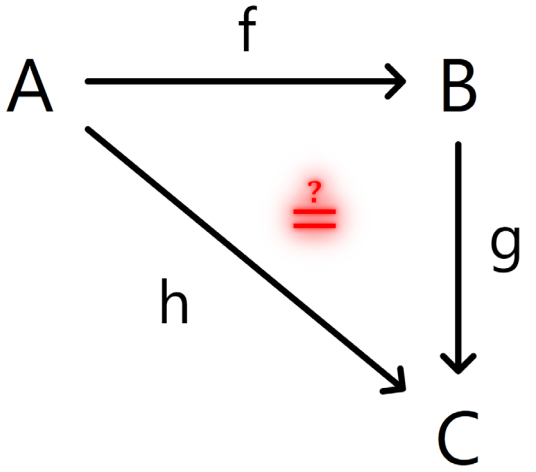
- A blob is a maximally connected set of index morphisms
- A cell is two lists of composable index morphisms, called left and right legs
  - Common index domain and codomain on legs
  - Cyclic cells currently not used
- Each blob is scanned to find all cells
- The blob on the right has three connected pullbacks and a total of 19 cells
  - 16 computed commutativity
    - Cell symbols normally turned off
  - Cell symbol auto-placed near barycenter of cell's index objects
  - Hovering over symbol shows morphisms in cell

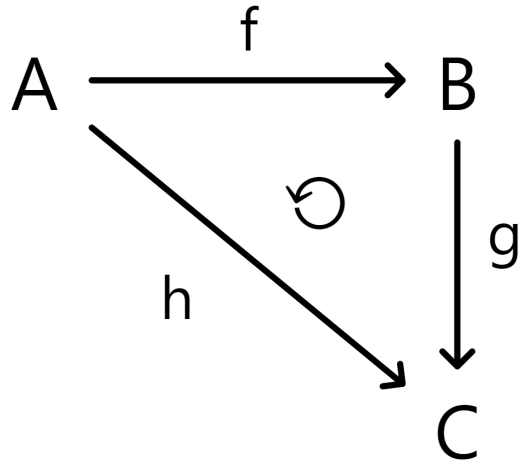


Assertions and Commutativity

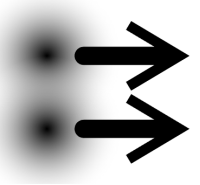


- Known equalities such as composites, identities, … are registered when the morphism is created
- Unknown cells are submitted to the equality engine
- If the engine cannot determine if the cell's two legs are equal, it returns an 'unknown' status as seen in the example
- The user may assert the status of the cell to be commuting
- The engine then recognizes elsewhere that h=gf when embedded in another blob or referencing diagram

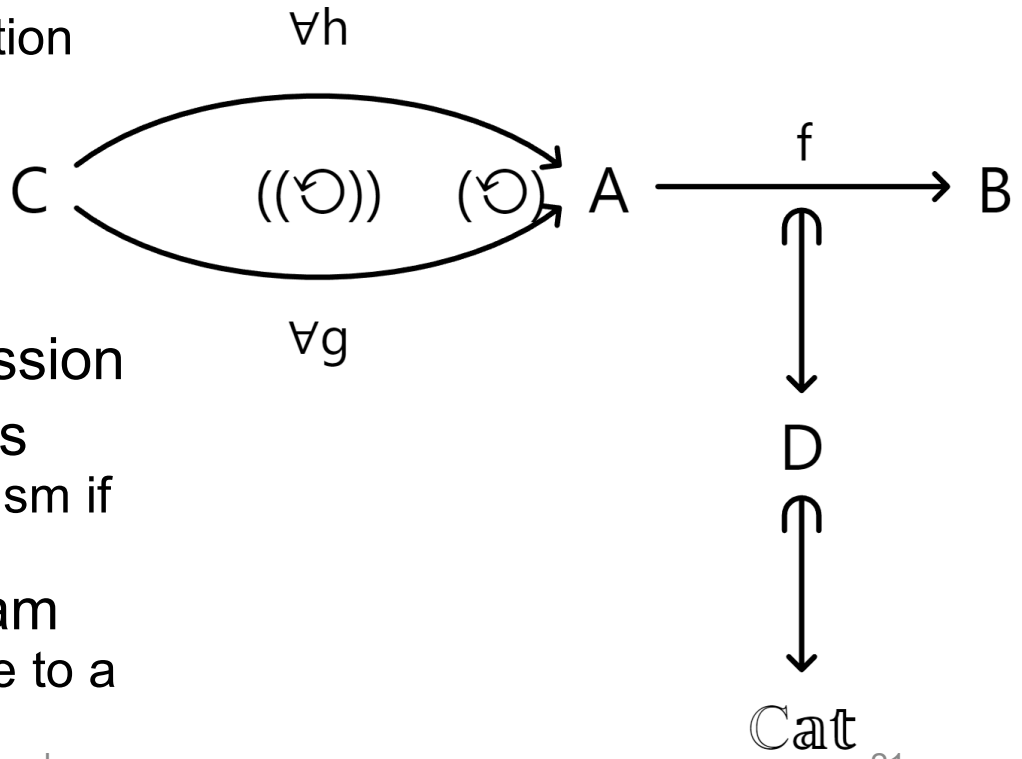




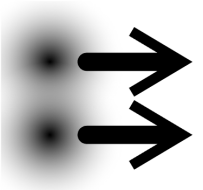
Definition of Definition



- A definition is a list of blobs
  - A sequence of morphisms is derived from the selected blobs
    - Said blobs are scanned for bare morphisms (not constructed from others)
    - The bare morphisms provide the definition's sequence, or required terms to start the definition
  - Any assertions in the blobs become part of the definition
- As needed, elements in the sequence may have quantifiers attached to be part of the definition
- Assertions and quantifiers may be reordered
- Parentheses are drawn around cell symbols and quantifiers to show depth of the term in the expression
- The definition of monomorphism on the right reads
  - For a category D in Cat, f:A→B in D is a monomorphism if for every g:C→A and h:C→A where fg=fh, then g=h
- A definition creates an object in the Actions diagram
  - Makes the definition accessible as an action available to a category



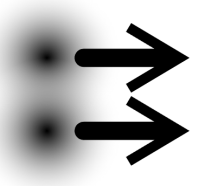
Definition as Template



- Some definitions have complex morphisms that can be generated from the definition's sequence
- If the user selects morphisms that match the sequence, then the user may generate one of the complex morphisms using the selected sequence in lieu of the definition's
- E.g., factorial defined with 64-bit floating point could be instantiated as 128-bit factorial by providing pattern matching 128-bit versions of the factorial's sequence
  - Select $=0_{128}$, $decr_{128}$, $mult_{128}$, and 1.0_{128} in the same sequence as the definition of factorial and action to instantiate appears

Definition Instance

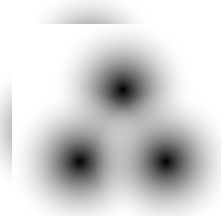
Let x be an X



- If a definition is available in a user's session, the definition may be instantiated in another diagram
  - Forces a reference back to the definition diagram
- Create a new sequence of morphisms that matches the pattern established by the definition's sequence for the current categorical context
- Definition instances starts theorems

Definition of Theorem

Let x be an X, then x is a Y

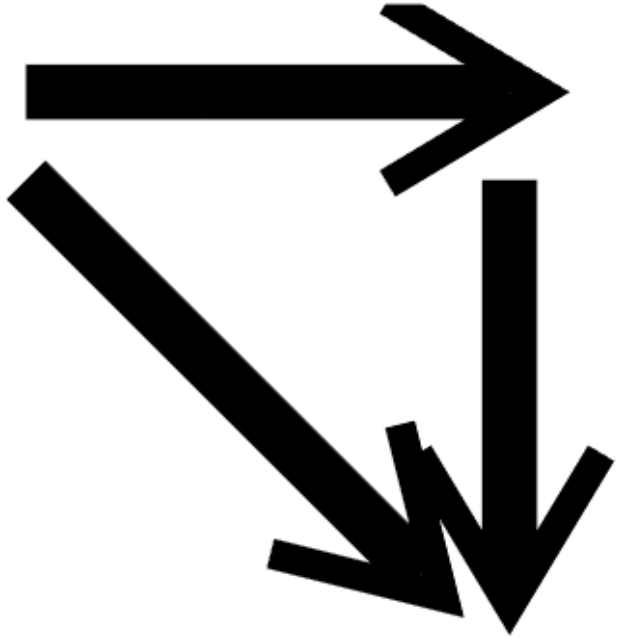


- A theorem has a source and a target
  - Source is a definition instance
  - Target is a definition
  - Effectively: Let x be an X. Show x is a Y.
- The definition instance's sequence is mapped by user to the target sequence
- Then the target definition's assertion cells are recreated using the substituted sequence
  - If equality engine determines all cells commute, theorem is good
    - If not, work on it
  - The definition instance's assertions can be generated to provide additional bridgework

C++

- For diagrams with C++ code, they are set to Set
  - Natural numbers, floating point, strings, …
- Attach C++ code to bare morphisms
  - No longer bare then
- Create morphisms with composition, …
  - Selected morphism becomes top level entry point
  - Use stdin/stdout morphisms to run from command line
- C++ code generator downloads morphism as a .cpp file
  - Tested with gcc

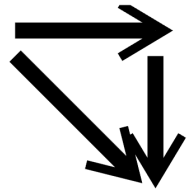
$$\mathbb{F}_{64} \times \mathbb{F}_{64} \xrightarrow[\texttt{\%2 = \%0 * \%1;}]{\text{mult}_{64}} \mathbb{F}_{64}$$



How Does Catecon Do It

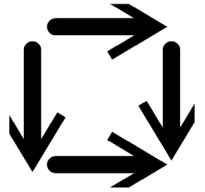
Architecture, Algorithms, Internals

Categories



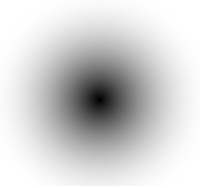
- In Catecon a category is a list of objects and morphisms
- Categories are not saved as a file
  - Diagrams are
- The entirety of the context of a category like Cat or Set is determined by all the reference diagrams available to the top diagram under consideration
- Index categories, the domain object for diagrams, provide the visualization of elements from categories like Cat or Set

Diagrams



- Diagrams are saved as .json files
  - E.g., https://catecon.net/diagram/hdole/category.json
- The codomain of a diagram is a category such as Cat or Set
  - Working scope changes based on selected elements
- The domain of a diagram is an index category
  - Responsible for visualization
  - Contains index objects, morphisms, and text
- JSON file may have nested categories, such as Cat with object D which is a category, which then lists its constituents used in the diagram

Index Objects

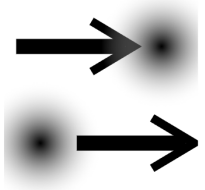


- Map to an element of a target category
- Has xy location on diagram web page
  - Catecon uses SVG to display the diagram
- An object's top-level factors outlined using the object graph
  - Double-click on a factor to create a projection/injection
- Maintains list of index morphisms that originate at the object
  - Dually list of index morphisms that terminate at the object
  - Cell assembly and morphism assembly use these to trace blobs
- Has set of cells that the index object belongs to

Index Morphisms

- Index morphisms map to morphisms in some target category
  - Index morphism domains and codomains only connect in same category
- Index morphisms are drawn according to the location of the index morphism's domain and codomain index objects
- Index morphisms may have an attached quantifier if the blob it is in uses it as a quantified element
- Has Bezier offset for bending the morphism appropriately
- If target morphism used as reference, then 'reference' attribute set
  - Appears dashed
  - Quantifiers and references do not mix in the same blob
- A 'flipName' attribute controls which side of the arrow the proper name appears

Actions



- Catecon has a series of actions that may invoke a construction depending on what is selected
  - https://catecon.net/d/hdole/actions
- Each action has a pattern that must be matched, e.g.
  - Select morphisms in domain/codomain sequence and composition action is shown
  - Select objects in same category then could create product, coproduct, hom objects
  - Select two morphisms with common codomain and pullback action is available
  - User adds their own actions by creating definitions

Naming Convention

- Unique name for all objects and morphisms
  - <username>/<diagram-name>/<base-name>
- The base name is a unique name in the diagram's context
- Separate proper names for objects and morphisms allows for appealing names such as f^{-1} versus finv
  - Proper names are not unique
- Each operator lists its elements in its base name
  - The composite of stdin to factorial to stdout has a base name of:

    'Cm{hdole/cpp/stdinToF64,fact,hdole/cpp/F64ToStdout}mC'

Signatures

- Different names across diagrams may refer to the same element
- To test for equivalency, Catecon uses signatures of objects and morphisms
  - Derived from SHA256
- For a bare object or morphism, the signature is the SHA256 hash of its name
- For a multi-object or morphism, the signature is the signature of its constituents' signatures, plus operator type and duality as needed
- A named object or morphism has the same signature as its base object or morphism
- Morphism signatures are registered with the equality engine
  - E.g., when f:A→B is instantiated, we know $f \circ 1_A = f$ and $1_B \circ f = f$
  - So, load legs [sig(f), sig(1_A)] and [sig(f)] as equivalent, and similarly for B

Multi-Objects and Morphisms

- In Catecon many objects and morphisms are constructed using a list of objects or morphisms
  - Composites
  - Products and coproducts
  - Hom
    - Only two
  - Product/Coproduct assemblies
- Scanning an expression is easy as it is just scanning the contained elements

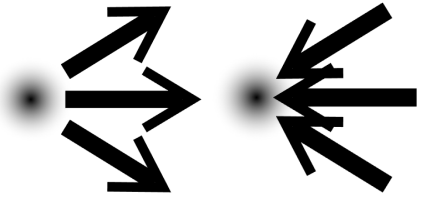
Factors (1)

- Core aspect of Catecon for accessing terms in an expression
- A factor is a list of integers
- Locates a term in an expression
- The empty list [] refers to the entire expression
- For example, the location of B in (A×B)+C is the list [0, 1]
  - 0 refers to the first term (A×B)
  - 1 refers to the second term in (A×B), or B
- In the morphism A→[B, A×B] the second B's factor is [1, 1, 1]
  - The first 1 refers to the codomain [B, A×B]
  - The next 1 refers to the second term in [B, A×B], or A×B
  - The next 1 refers to the second term in A×B, or B

Factors (2)

- Objects, morphisms and string graphs have a getFactor() method to give the sub-component from a given factor
- Scanning an object, morphism, or graph maintains a tracking factor
  - Starting a scan has tracking factor set to [] to represent entire element
  - As scanner dives into a constituent element, the index of that element is pushed to the end of the factor
    - E.g., scanning (A×B)×C first hits A×B as the first constituent so the tracking factor becomes [0]
    - Next the A is scanned, another 0 is pushed onto the factor to yield [0, 0] as the position of the A in the expression
    - Popped on return

Factor Morphisms

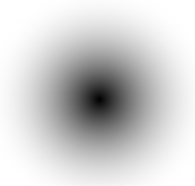


- As noted earlier, a factor morphism is an object with a list of its terms
- Those terms are described as factors indexing their location
- The projection (A×B)×C→A×B is described by the object (A×B)×C and the factor [0] as A×B as the first factor in (A×B)×C
- The morphism A×B→A×B×1 is described by the object A×B and the factors [0], [1] and [-1], where [0] is A, [1] is B, and the terminal object 1 is given by [-1]
- The twist morphism A×B→B×A is given by the object A×B and the factors [1], [0]

Lambda Morphisms

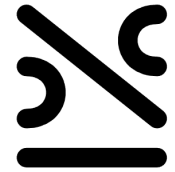
- A lambda morphism is described by taking a morphism, and then stating which factors from its domain or codomain's hom's domains, end up in either of the two for the curried morphism
  - The domain factor list gives those pre-curry factors remaining in the domain after currying
  - The hom factor list gives those pre-curry factors residing in the codomain's domain side of the [,] hom sets

Graph of an Object



- The graph of an object is created on demand
- Graph has a list of sub-graphs
- Maintains width and position
  - Width represents the width of the text for that node
  - Position is the offset from the left of the entire object display string to the middle of the text for that node
    - This gives the horizontal location of the string graph's links
    - Vertical location given by the index object being graphed
- Each operator in target object's expression states if it needs parentheses or not
  - Product and coproduct: Yes; Hom: No
  - Allows for additional offset of parentheses in the expression
- Walking the hierarchy of the target object generates the object graph and all positions

Graph of a Morphism

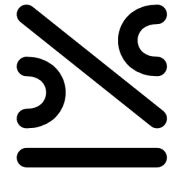


- The top-level node for the graph of a morphism has two graphs, one for the domain object and one for the codomain object
  - Thus, the morphism's graph factor [0] refers to the domain object graph, and factor [1] refers to the codomain object graph
- Connections between leaves in the morphism graph are called links
  - A link is a factor giving the location of another leaf in the graph
  - The leaf node has a list of links
  - The factor gives the location in the morphism graph for the other end of the link
- From a link's start and end point, a Bezier curve is created to attach the two visually
- For vernacular code generation, a morphism's full internal graph can be used
  - E.g., composite has object graphs and links for all intermediate objects and morphisms
  - Strings on the full graph are mapped to variables in the C++ code generator

Graphs of Factor, Product, Hom, Lambda Morphisms

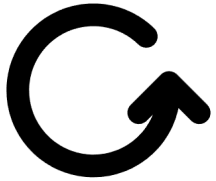
- To compute the graph, each morphism type must construct its links from its underlying information, be it factors or morphisms
  - Factor morphism: The list of factors gives the identity linkages between the domain and codomain
  - Product, coproduct, and hom morphisms: The links from the constituent morphisms are copied into the appropriate slot
    - Copy links means pushing the index of the constituent morphism to the front of all the links factors
  - Lambda morphism: The pre-curry morphism has its links copied to the lambda morphism with the dom factors giving the link copying instructions for the domain, and the hom factors gives it for the D's in $[D_0, [D_1, [D_2, [\ldots, B]\ldots]$
    - Copying links means reindexing them by the dom and hom factors so they move to the correct locations

Graph of a Composite



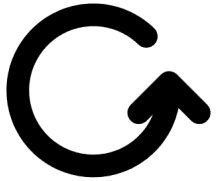
- For a composite morphism, there's more work than just copying reindexed links
- As with most of the constructions, first step is to form the graphs of its constituent morphisms
- For a composite, A→B→C→D, the graph of A×B×C×D is formed (called the sequence graph)
- Next each constituent morphism graph of A→B→C→D is merged into the sequence graph
  - B and C get merged twice as there are two of them in A→B, B→C, C→D
  - Links are reindexed from starting with either 0 or 1 to the offsets in the sequence graph (0, 1, 2, 3 in the A×B×C×D example)
- The sequence graph now has a lot of links
  - Used by code generator
- Next trace the links
  - For a leaf node in the sequence graph, visit each link (aka factor)
    - Use getFactor(link) on the sequence graph to get the linked-to leaf node
    - Visit its links and keep going until all connected links have been visited
    - Record these traced links in the sequence graph
- We only want links between the first and last graphs or themselves in the sequence graph
  - In the example A→B→C→D any links between A and B or A and C are removed
  - Similarly remove links between D and B or C
- The first and last graphs in the sequence are copied to be the final domain and codomain graphs of the composite
  - The domain graph is a simple copy with outbound links reindexed to 1 (versus 3 in the A→B→C→D example)
  - The codomain graph is a simple copy with its internal links reindexed to 1

Cells in a Diagram



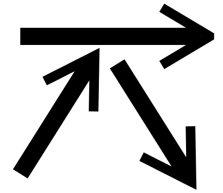
- A diagram's blobs and cells are determined when the diagram is loaded
- Cells with assertions are carried in the index category
- Such a cell is identified by the names of the index morphisms on its legs
- At load time, if a cell has an assertion, it is registered with the equality engine
- If an index morphism on a cell has a universal quantifier attached, then whenever a morphism is created that matches that pattern, the cell with that pattern is then registered with the equality engine

Equality Engine

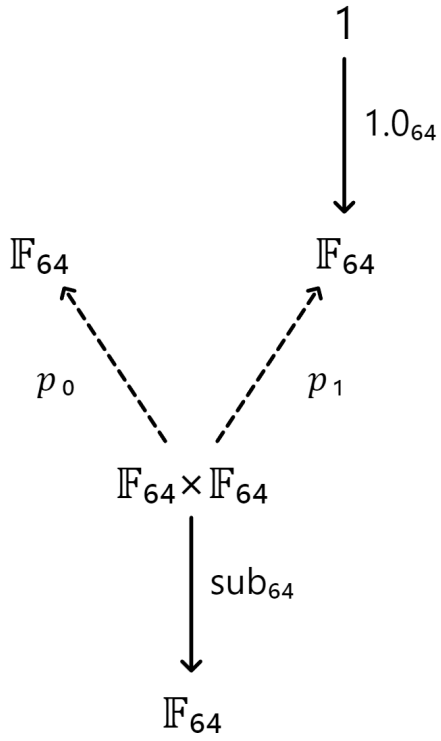


- This feature determines commutativity of cells in a diagram
  - Runs as web worker
- Knows nothing of morphisms
  - Legs are registered as an array of signatures
- An asserted cell sets its two legs as signatures to be equivalent in the engine
- A map from elements names to signatures gives tracking
  - Editing means having to remove assertions from the engine
- Identities are tracked
- For cells with unspecified commutativity, the engine scans portions of the cells sub-legs to find possible substitutions

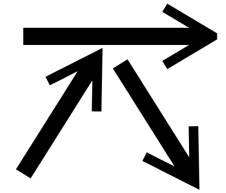
Morphism Assembly (1)



- Building a complex morphism from individual steps of compositions, products, and so in is a lot of work
- Instead form a diagrammatic language to simplify assembly
  - Specify certain index morphisms as reference morphisms (dashed)
  - Candidate target morphisms: Single factor projections or injections, identities
- For a reference morphism, think of the arrow's functional flow as being reversed

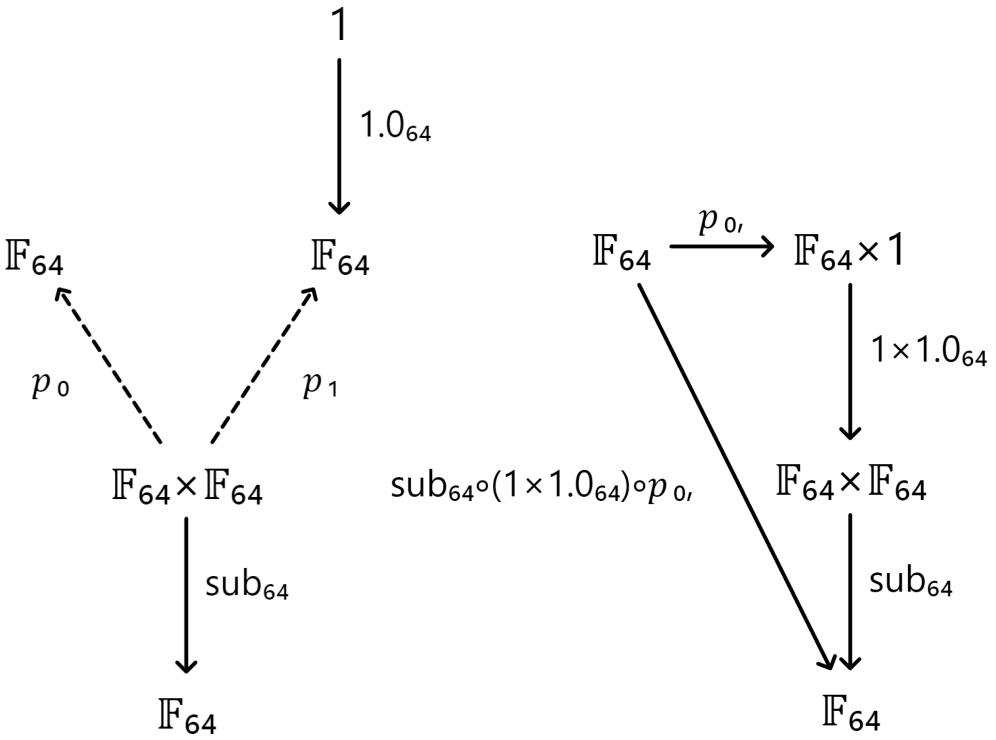


Morphism Assembly (2)

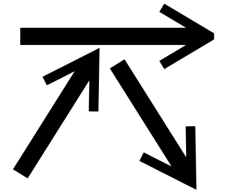


Decrement by One

- Both $\mathbb{F}_{64}$ terms of the sub_{64}'s domain are projected and turned into references
- The left-hand projection p_0 means wait for input
- The right-hand projection's p_1 reference is satisfied by the 1.0_{64}
- The assembly of the blob is shown on the right

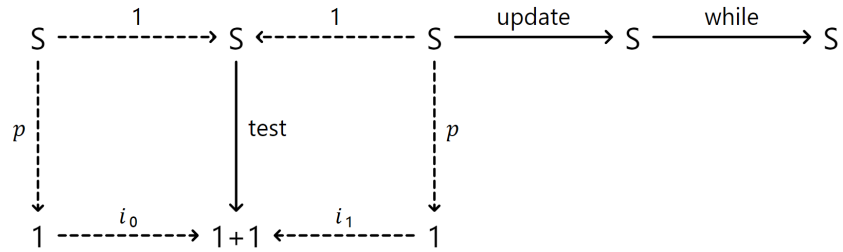


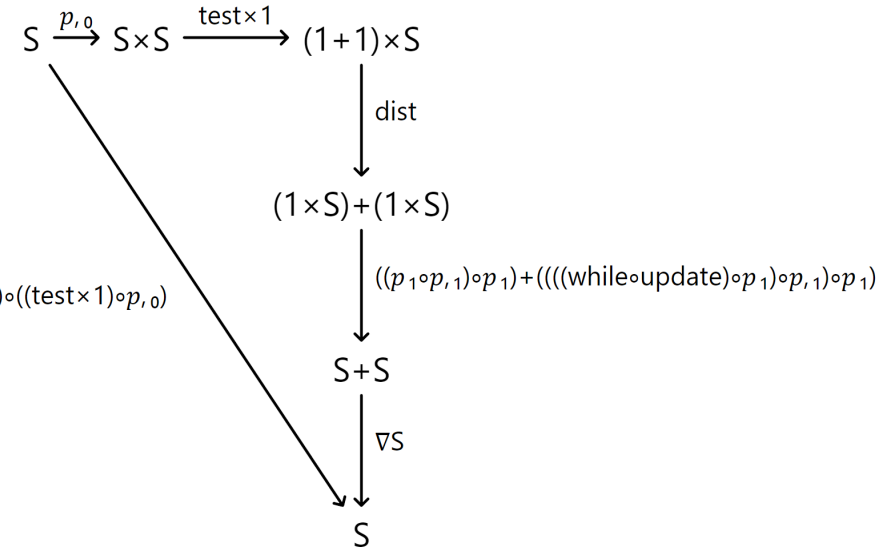
Morphism Assembly (3)



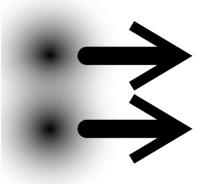
While

- The vernacular programming notion of ‘while’ is not built-in, so we define one with references and recursion
  - Given an object S representing our state, apply the test morphism S→1+1
  - The first ‘1’ from i_0 represents false, and i_1 represents true
- The injections i_0 and i_1 as references represent a cover of the base 1+1
- When the codomain of one of the cover’s elements receives a hit, then morphisms attached to the cover element’s domain are run
- In the case of false, the reference projection p:S→1 on the left is run
  - For that S to be defined, its state cannot come from S→1
  - So, the upper left reference identity S→S then finalizes the state, which is no change
  - Hence when the test is false the state does not change
- In case the test return true, the i_1 reference is invoked, and then the S→1 on the right side
- Lastly, the identity reference S→S says to use the starting state and run it through the update morphism, following by the while morphism
- Assembling this blob produces the rather complex looking morphism that uses various factor morphisms, folds and distributions
  - Last step is to set the recursor for while to be that generated morphism
- In general, the base object for the cover need not be a coproduct
  - For example, the notion of ‘select’ on the object Str of strings is implemented by having each ‘case’ be a referenced element of Str, and the ‘default’ case covered by a reference identity
- Distribution is used to send all inputs to each separate composite path started by the cover
  - Only used factors of the input then projected to start each cover element’s composite path
- All composite paths starting from a cover must reconverge
  - That’s the final fold you see ∇S:S+S→S



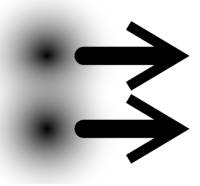


Definition



- Catecon keeps track of a definition in a diagram as a derived index text object
- Loading a diagram with a definition creates a corresponding user action in the Actions diagram
  - Appears as object in Actions
  - It is this action that scans the selected set to see if the action can be invoked for the definition

Definition Instance

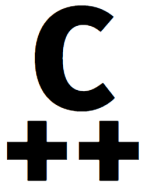


- An instance of a definition in a diagram is a derived index text object
- Creates reference to diagram holding the definition, if required
- Contains a sequence of elements in a target category that correspond to the definition’s sequence
  - Instance’s sequence must satisfy pattern of definition’s sequence

Theorem

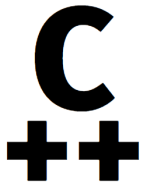
- A theorem in a diagram is a derived index text object
- A theorem tracks all the cell assertions that were generated when the source definition instance was bound to the target definition
- A morphism in the Actions diagram is created to record the theorem from the source definition to the target definition

Code Generation (1)



- For a selected morphism, the C++ code generator stitches together the C++ code from the leaf morphisms of the given morphism's expression tree
- The full internal string graph of the morphism is constructed
  - The links in each individual morphism's graph can be traced the entirety of the given one
  - A string of links represents same influence or confluence
  - Each string is assigned its own unique variable name
  - Domain and codomain terms are constructed into a function's arguments with assigned variable name from the attached string
- An entry point for the selected morphism is created
- Due to the full expansion of the selected morphism's graph, no other function entry points are defined
  - Recursion gets an entry point

Code Generation (2)



- For the factorial example, the bulk of the emitted code is shown
- The signatures of the morphisms are used as the function name
- In green, you see the recursive factorial function, along with its two invocations
  - One recursively
  - Other top-level entry point
- In this case the given morphism has stdin for domain and stdout for codomain
  - Reads the correct number of parameters on stdin
  - Prints result to stdout
- None of the constructed morphisms used in the factorial generated function calls

```
// hdole/Floats/F64 --> hdole/Floats/F64
//
void fun_3F82037F6F2be1de4fca3175ae74ac7179683323c2e015aea8d7164d5bbf3868(const hdole_Floats_F64 var_2, hdole_Floats_F64 & var_3)
{
    hdole_Floats_CPo_Br__n_1_c__n_1_rB_oPC var_4;
    hdole_Floats_F64 var_5;
    var_5 = 0.0;
    var_4 = var_2 == var_5;
    if (var_4)
    {
        var_3 = 1.0;
    }
    else
    {
        hdole_Floats_F64 var_6;
        hdole_Floats_F64 var_7;
        hdole_Floats_F64 var_8;
        var_8 = 1.0;
        var_7 = var_2 - var_8;
        fun_3F82037F6F2be1de4fca3175ae74ac7179683323c2e015aea8d7164d5bbf3868(var_7, var_6);
        var_3 = var_2 * var_6;
    }
}
//
// Composite
// hdole/Factorial/Cm(hdole/cpp/stdinToF64,fact,hdole/cpp/F64ToStdout)mC
// hdole/cpp/stdin --> hdole/cpp/stdout
//
void fun_37e66ac7b9e52326c0f779dbf081b0ea8a04a65a7b161d01c69b0197a26532695()
{
    hdole_Floats_F64 var_0;
    hdole_Floats_F64 var_1;
    std::cin >> var_0;
    fun_3F82037F6F2be1de4fca3175ae74ac7179683323c2e015aea8d7164d5bbf3868(var_0, var_1);
    std::cout << var_1;
}
int main(int argc, char ** argv)
{
    try
    {
        if (argc == 2 && (strcmp("-h", argv[1]) || strcmp("--help", argv[1])))
        {
            std::cout << "Diagram: hdole/Factorial" << std::endl;
            std::cout << "Morphism: hdole/Factorial/Cm(hdole/cpp/stdinToF64,fact,hdole/cpp/F64ToStdout)mC" << std::endl;
            return 1;
        }
        fun_37e66ac7b9e52326c0f779dbf081b0ea8a04a65a7b161d01c69b0197a26532695();
        return 0;
    }
    catch(std::exception x)
    {
        std::cerr << "An error occurred" << std::endl;
        return 1;
    }
}
```

Network Architecture

- Catecon.net is root level server
- Lower-level servers feed up through a chain of servers
- Diagrams on a given server are visible to all users of that server and all sub-servers
- Diagram .json and .png files are statically downloaded to web server’s indexeddb local storage

Server Architecture

- Local server can be setup on an individual's machine
  - WSL on Windows
- Requires Linux, node.js, MySQL
  - Javascript for node.js express web server
  - MySQL database for tracking diagrams
- User authentication not handled by local server
  - AWS Coginito handles signup/logins
  - Ensures global namespace is unique since usernames are unique
- All cookies from AWS Cognito
- For users connected to a local server, diagram edits are autosaved to the server rather than left in web browser's memory

catecon.net

Harry Dole