Categories for the Working Hacker

Philip Wadler
University of Edinburgh & IOHK
Lambda Days
22 February 2018

Smart Contracts

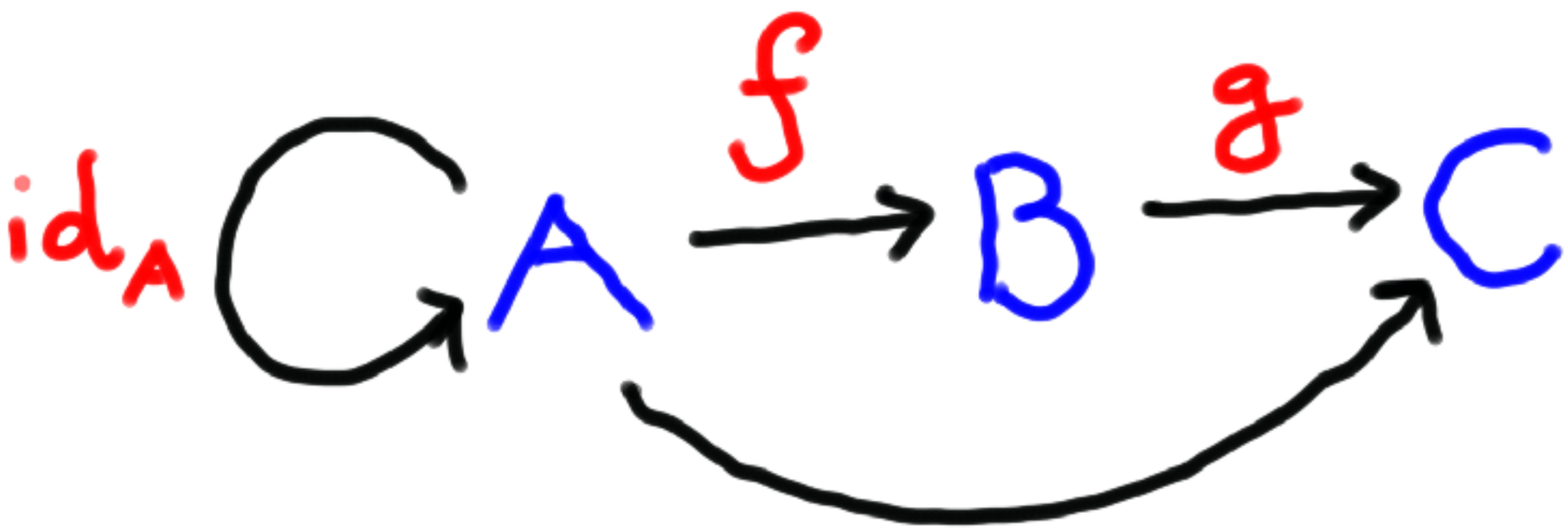
Simplicity

Michelson

Plutus

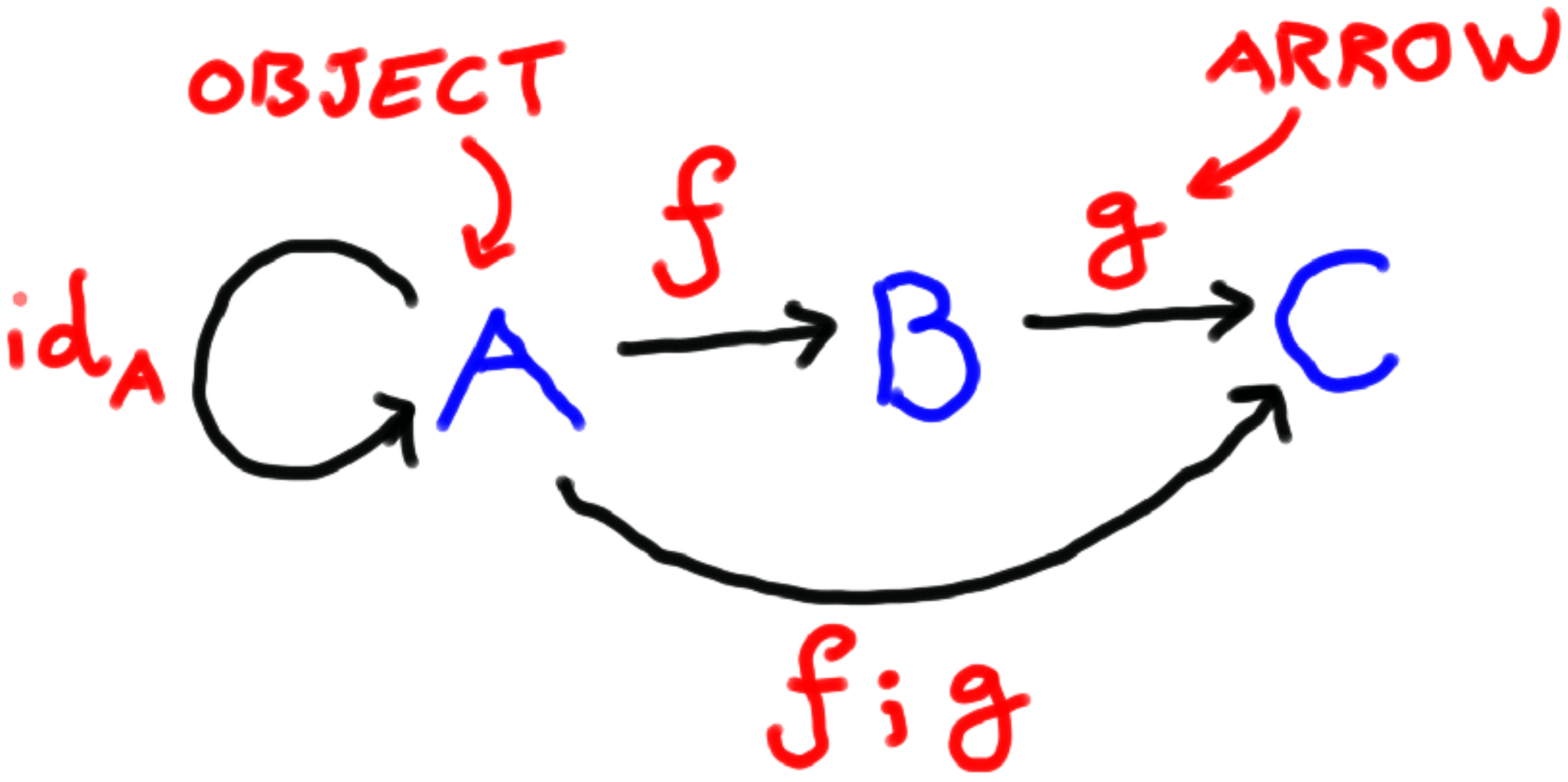
Categories

CATEGORIES

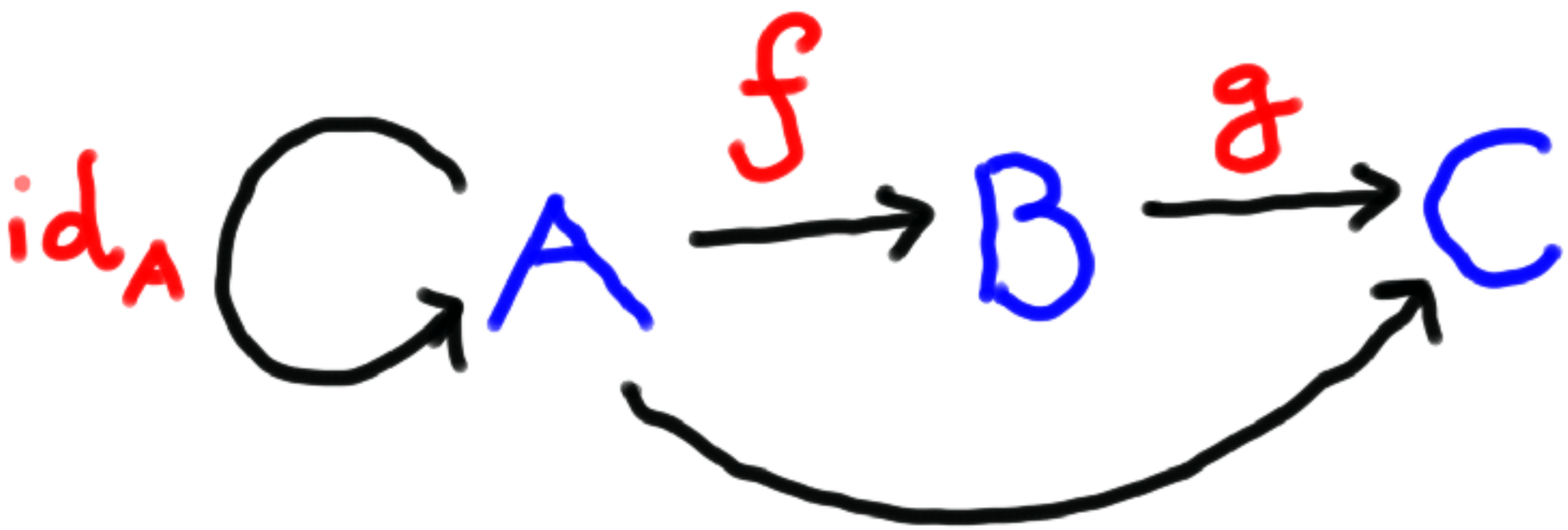


f ; g

CATEGORIES



CATEGORIES

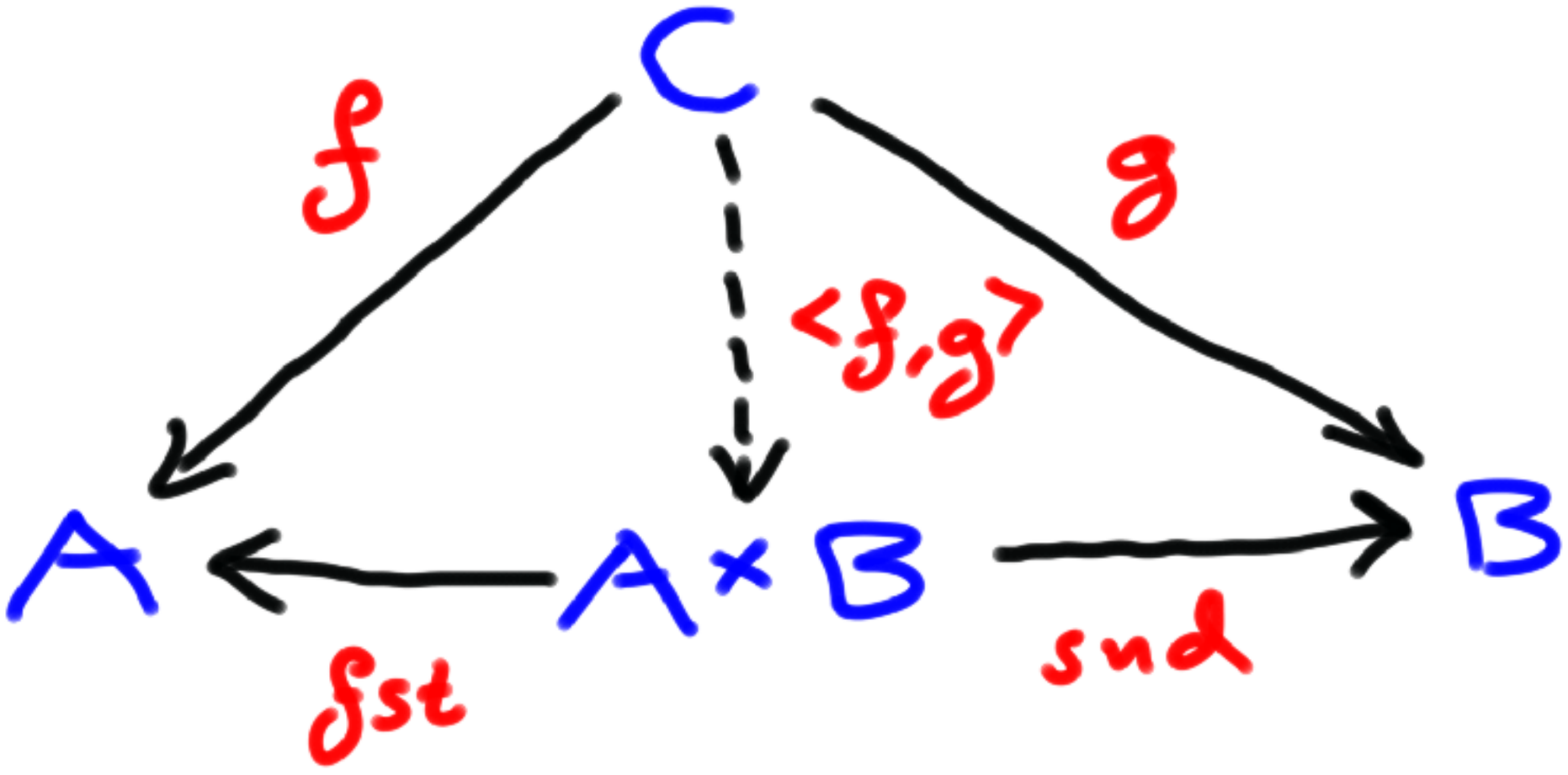


$$id_A ; f = f = f ; id_B$$

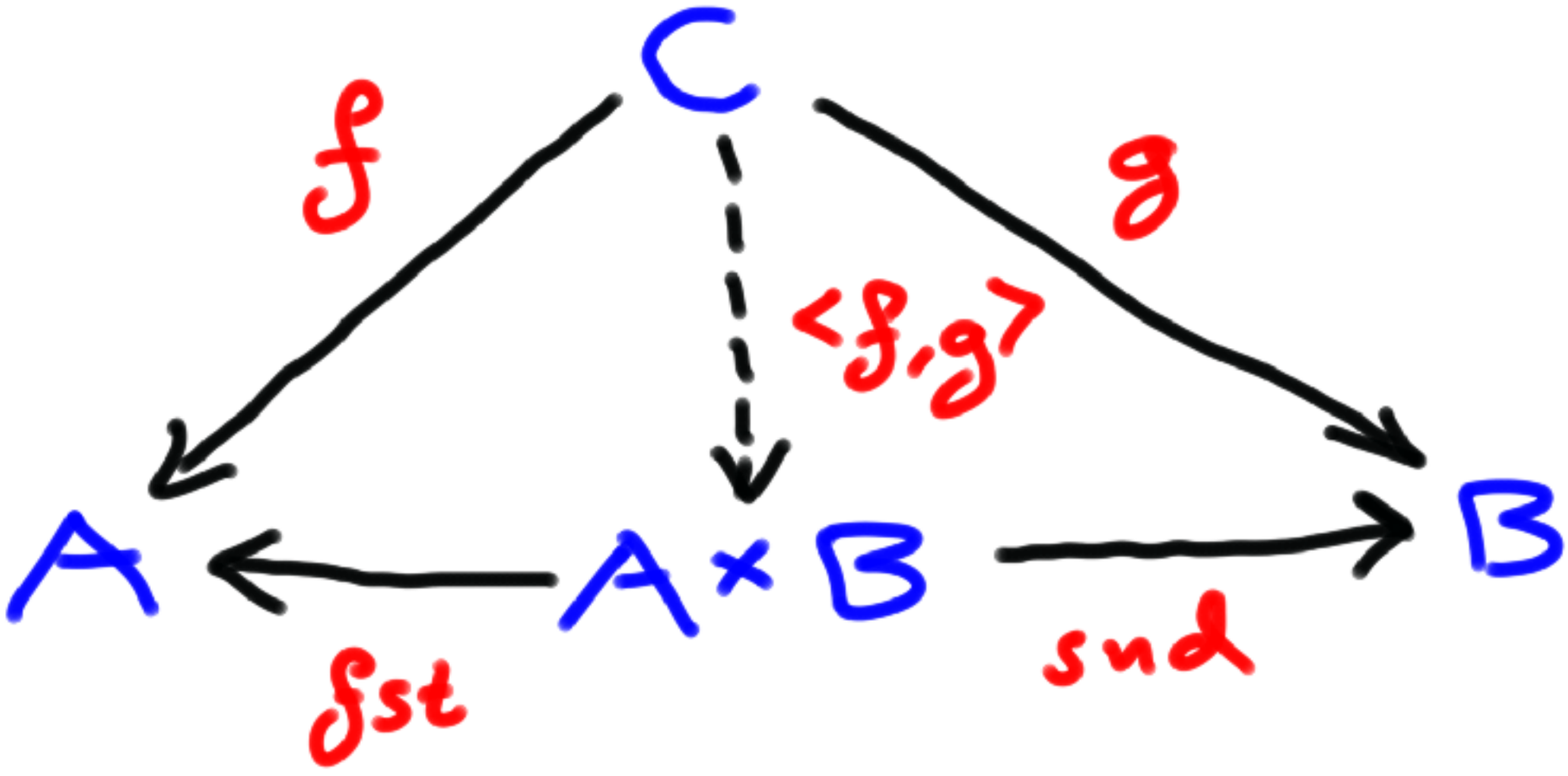
$$(f ; g) ; h = f ; (g ; h)$$

Products

PRODUCTS

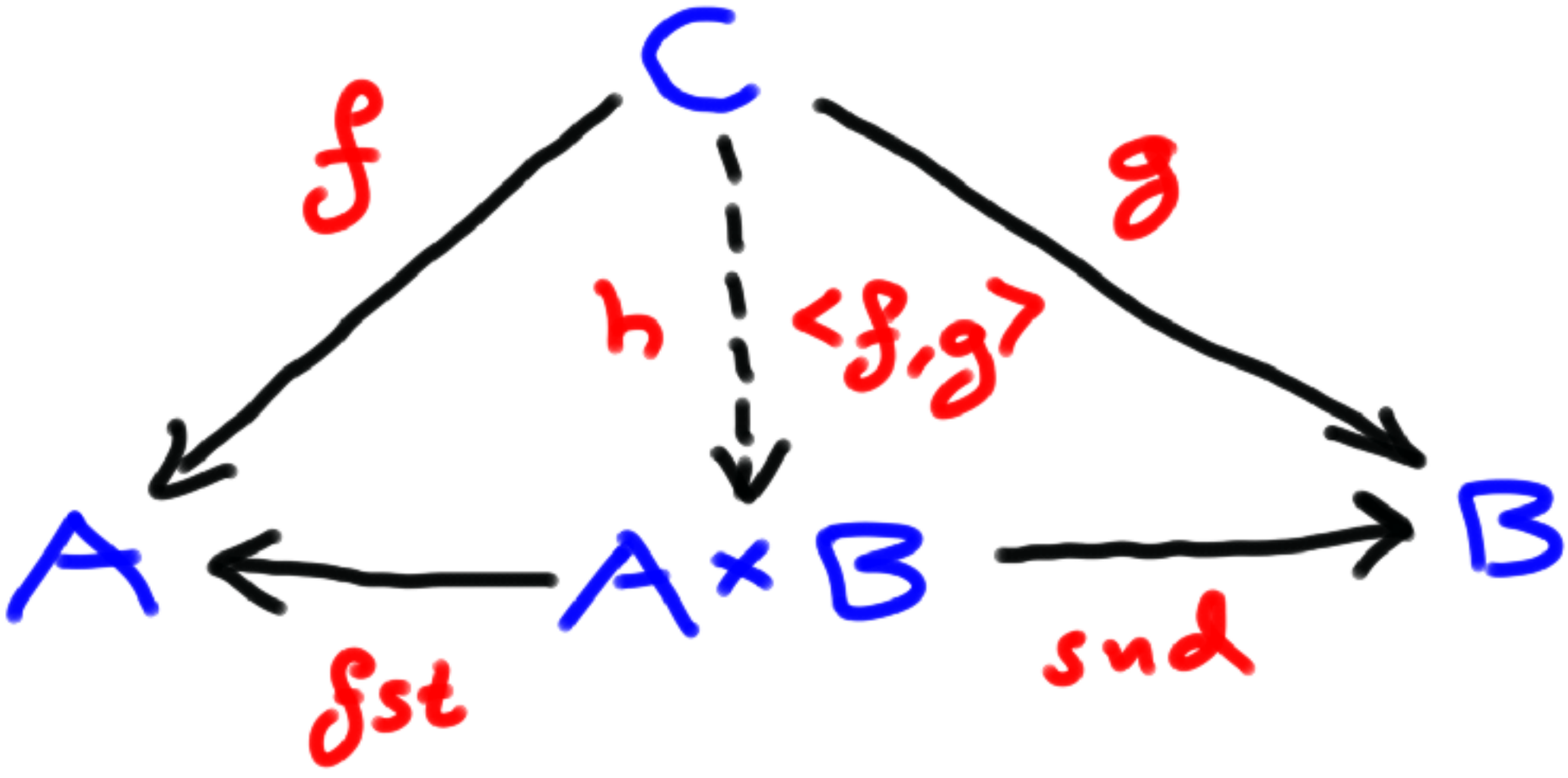


PRODUCTS



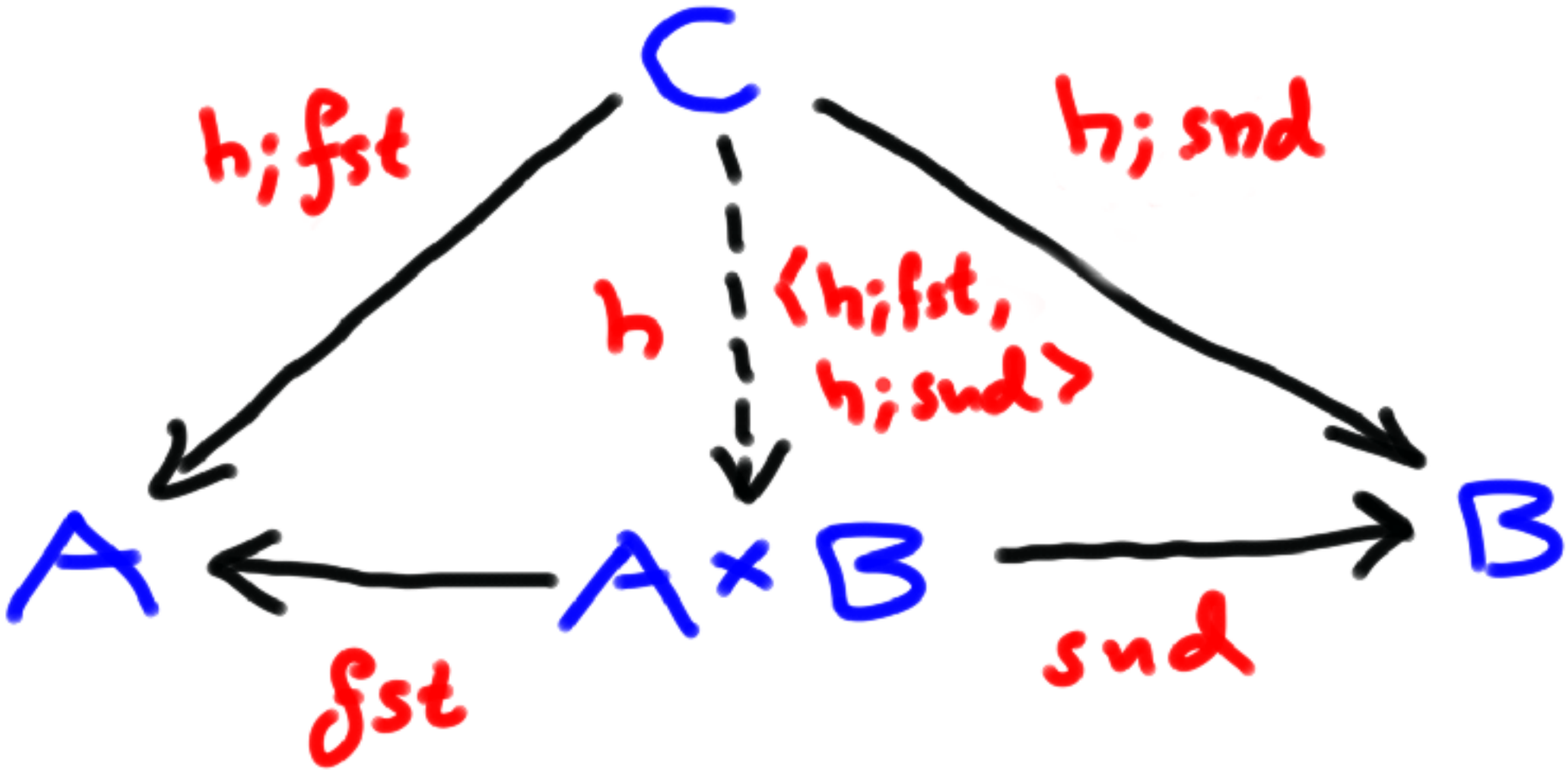
$$\langle f, g \rangle ; fst = f$$
$$\langle f, g \rangle ; snd = g$$

PRODUCTS



$$\frac{h ; fst = f \qquad h ; snd = g}{h = \langle f, g \rangle}$$

PRODUCTS



$$\frac{h;fst = h;fst \qquad h;snd = h;snd}{h = \langle h;fst\ ,\ h;snd \rangle}$$

PRODUCTS

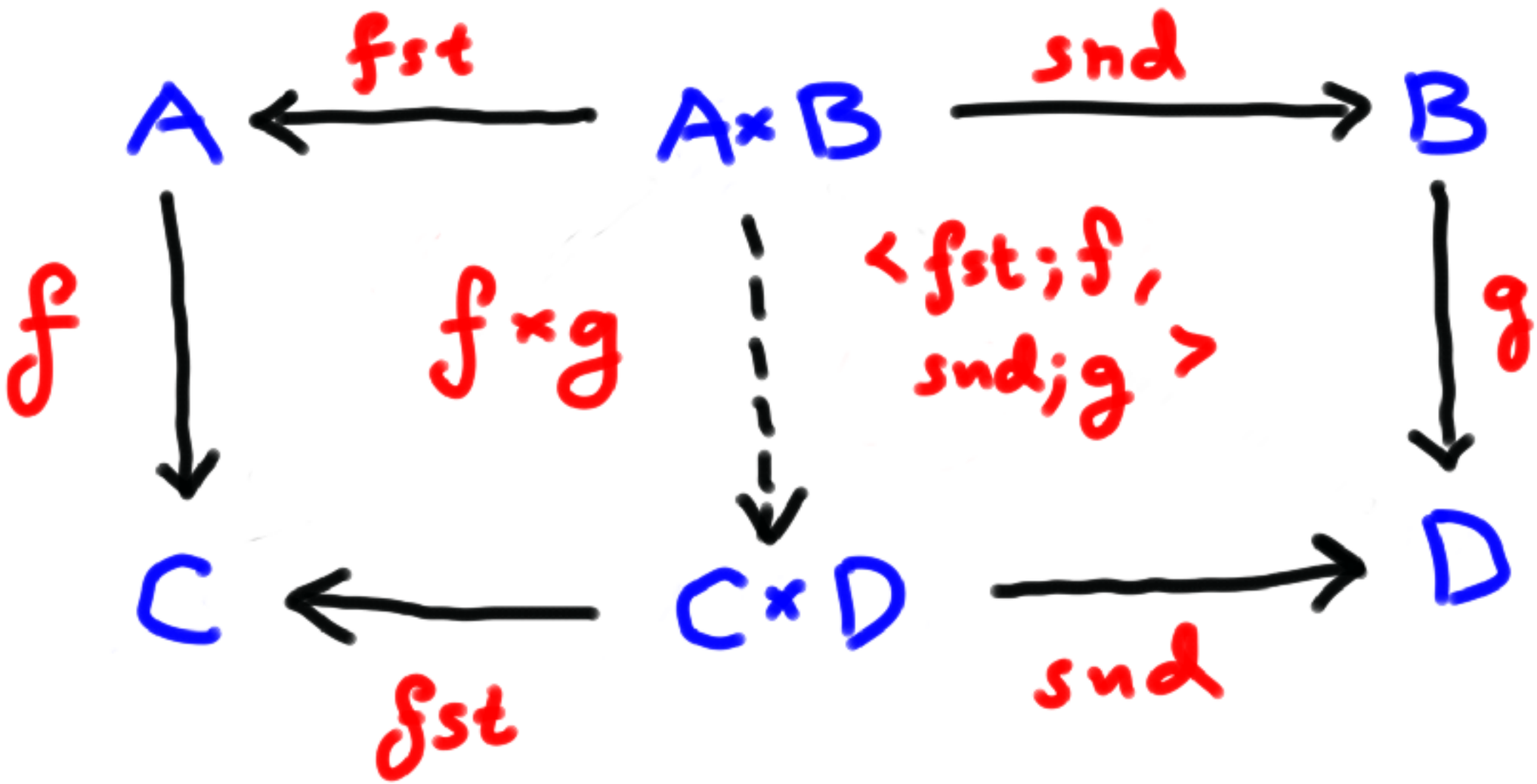
3 × 2

('a', 0) ('a', 1)

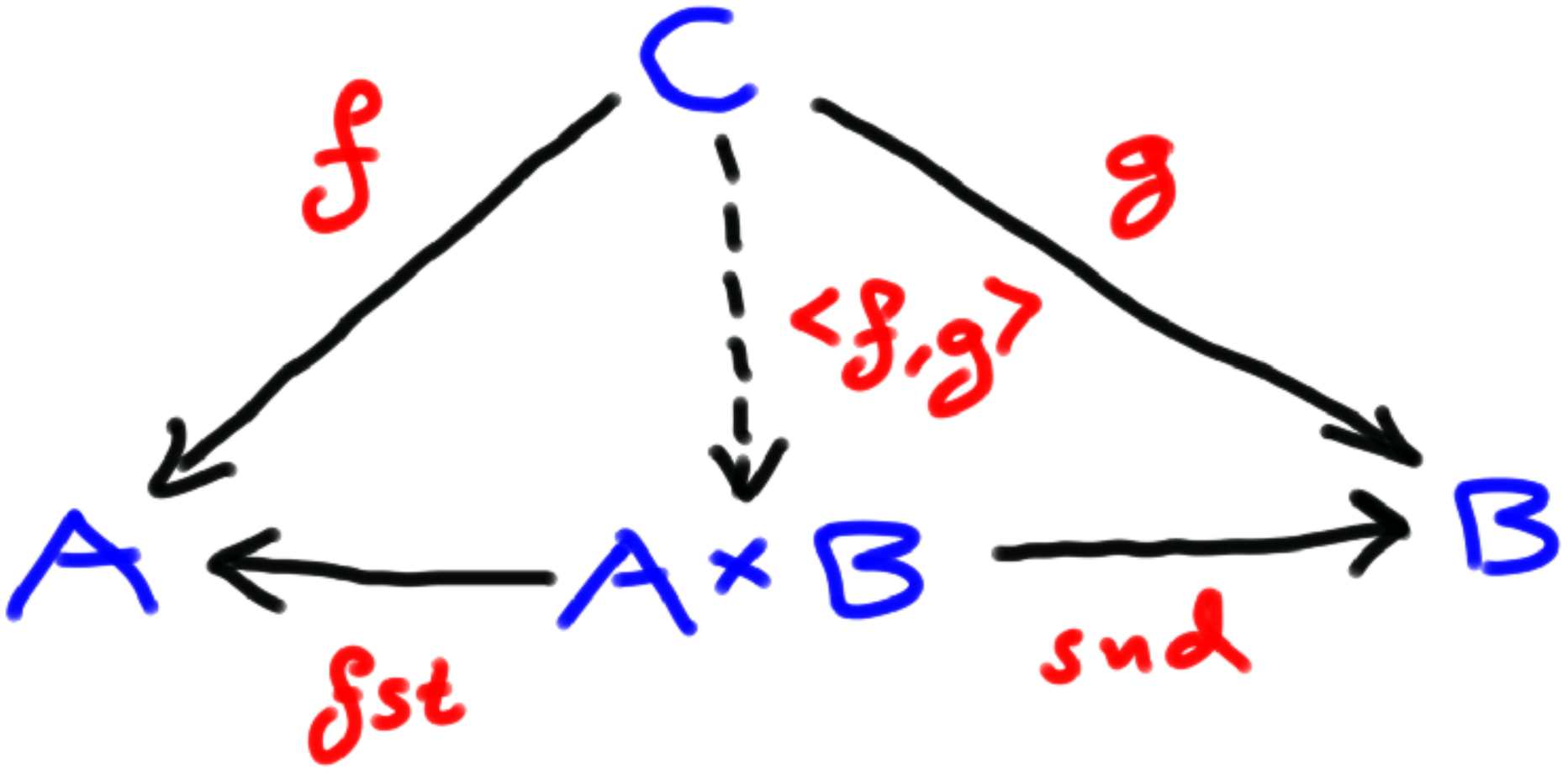
('b', 0) ('b', 1)

('c', 0) ('c', 1)

PRODUCTS



PRODUCTS



$$\mathcal{C}(C, A \times B) \cong \mathcal{C}(C, A) \times \mathcal{C}(C, B)$$

Products in Java

```
public class Product<A,B> {
  private A fst;
  private B snd;
  public Product(A fst, B snd) {
    this.fst = fst; this.snd = snd;
  }
  public A getFst() {
    return this.fst;
  }
  public B getSnd() {
    return this.snd;
  }
}
```

Products in Java

```
public class Test {
  public Product<Integer,String> pair =
    new Product(1, “two”);
  public Integer one = pair.getFst();
  public String two = pair.getSnd();
}
```

Products in Haskell

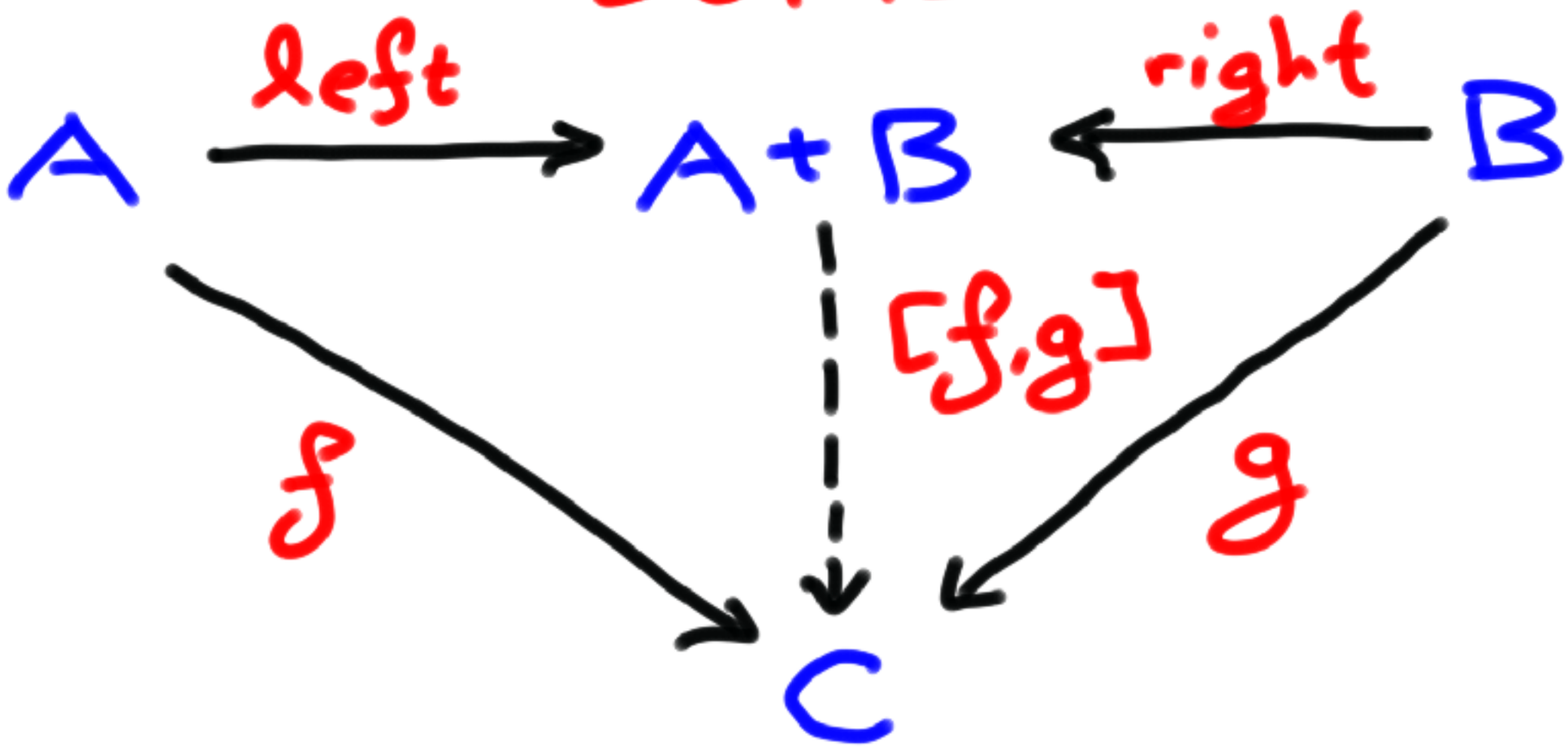
```
data Product a b =
  Pair { fst :: a, snd :: b }
```

Products in Haskell

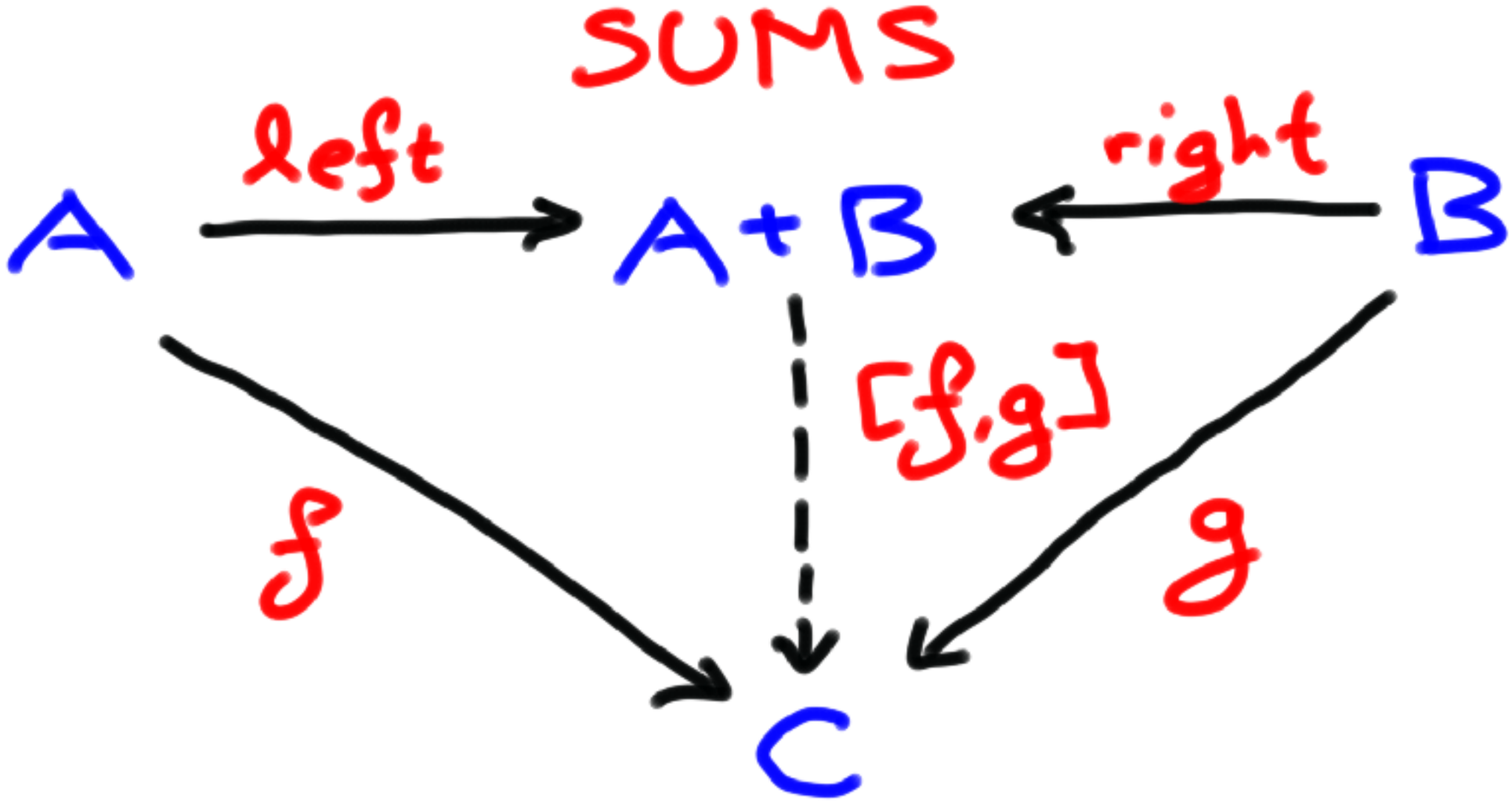
```
pair :: Product Int String
pair =  Pair 1 “two”

one :: Int
one =  fst pair

two :: String
two =  snd pair
```

Sums

SUMS

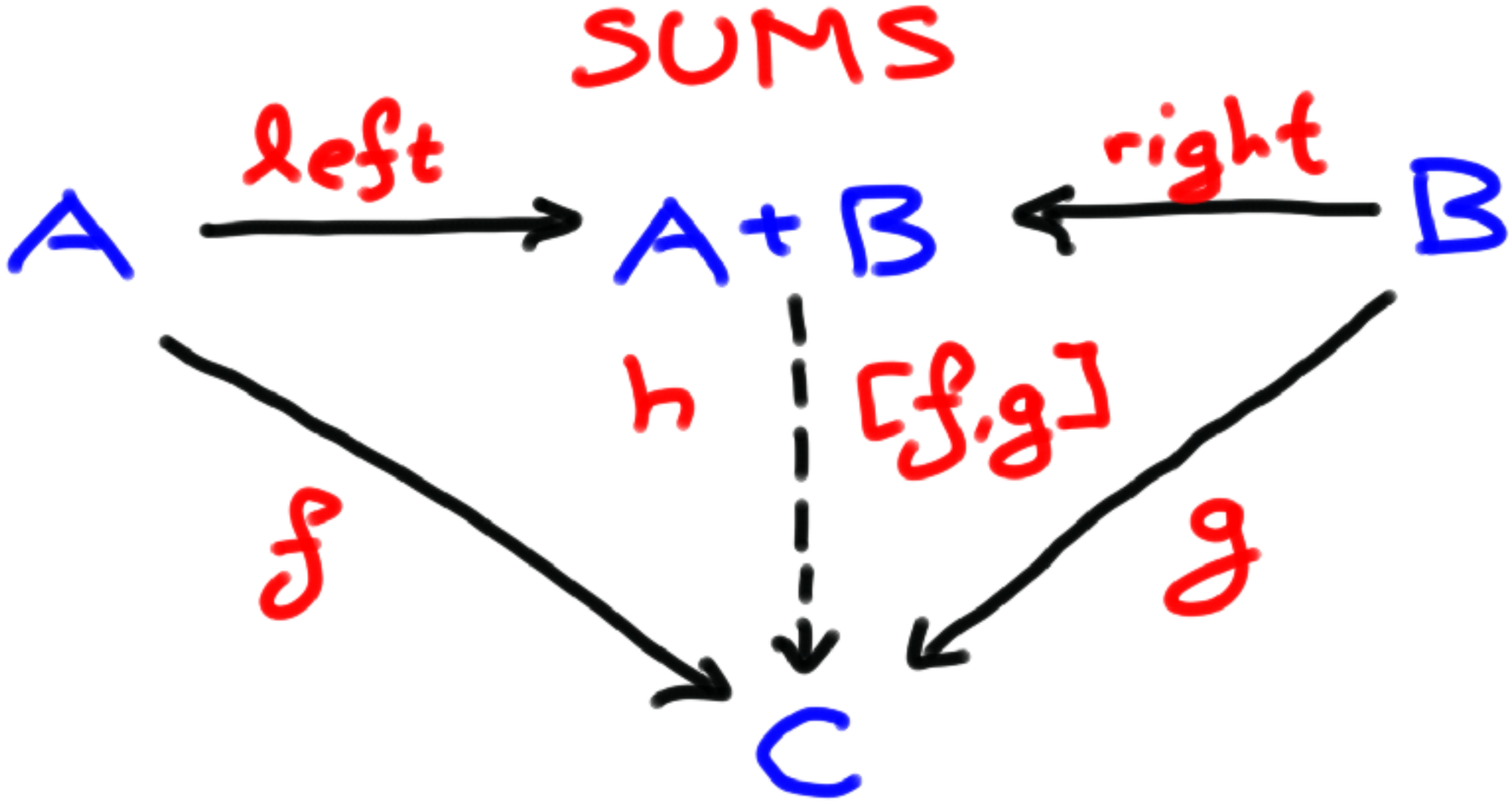


SUMS



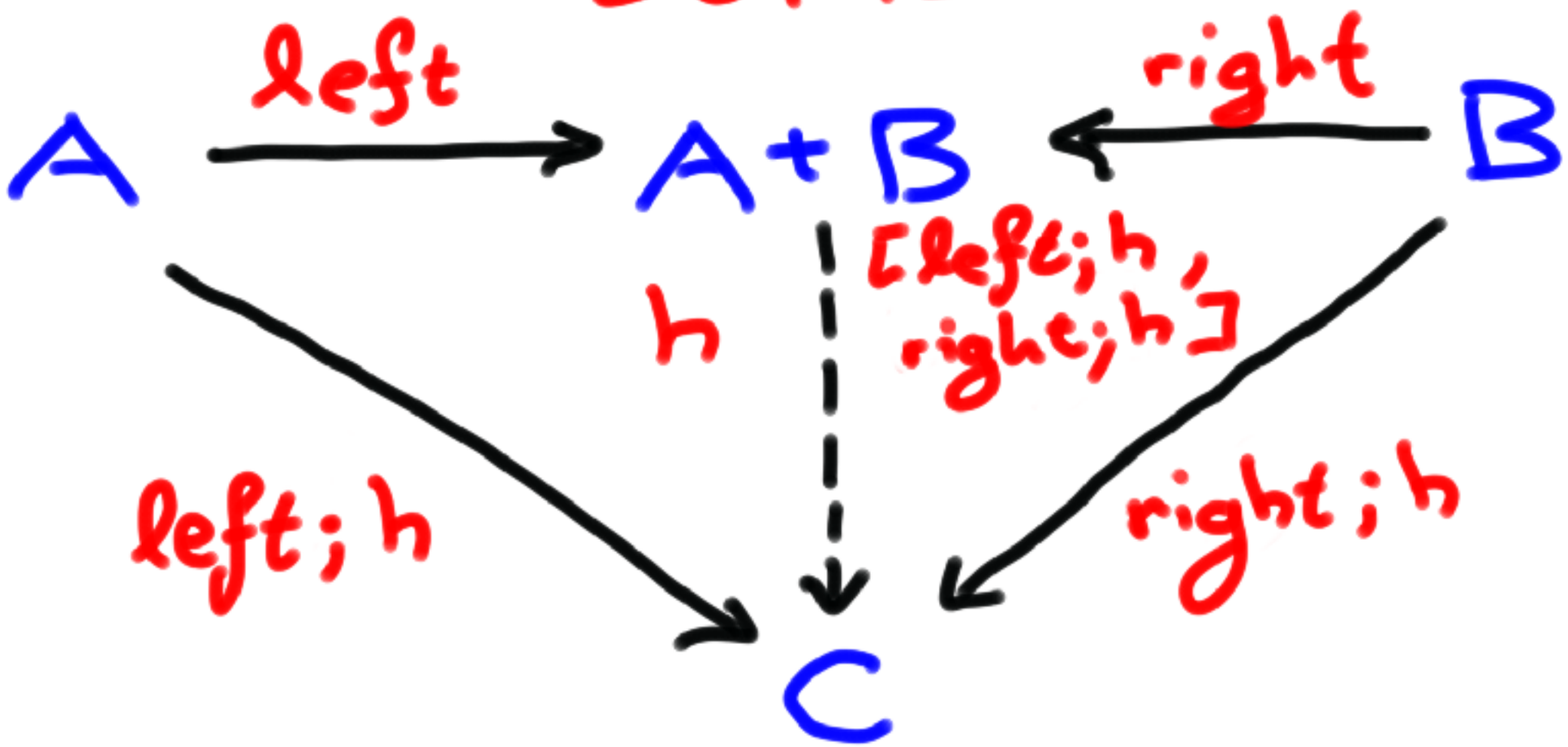
$$left ; [f,g] = f$$
$$right ; [f,g] = g$$

SUMS



$$\frac{left; h = f \qquad right; h = g}{h = [f, g]}$$

SUMS



$$\frac{left;h = left;h \qquad right;h = right;h}{h = [left;h,\ right;h]}$$

SUMS

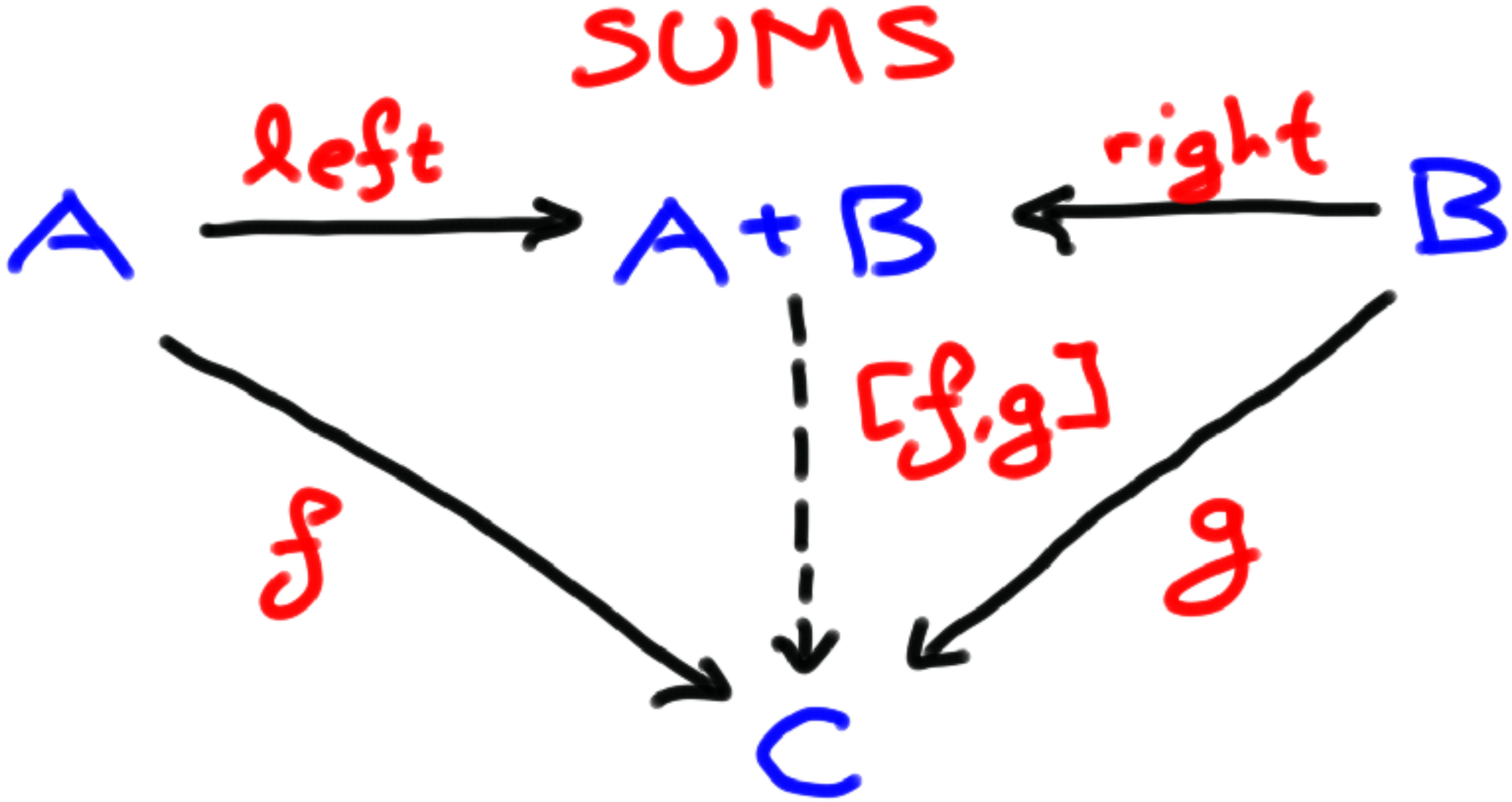
$3+2$

left 'a' right 0

left 'b' right 1

left 'c'

SUMS



$$\mathcal{C}(A+B, C) \cong \mathcal{C}(A, C) \times \mathcal{C}(B, C)$$

Sums in Java

```
public interface Sum<A,B> {
  public <C> C caseExpr(Function<A,C> f,
                        Function<B,C> g);
}
public class Left<A,B> implements Sum<A,B> {
  private A x;
  public Left(A x) { this.x = x; }
  public <C> C caseExpr(Function<A,C> f,
                        Function<B,C> g) {
    return f.apply(x);
  }
}
public class Right<A,B> implements Sum<A,B> {
  private B y;
  public Right(B y) { this.y = y; }
  public <C> C caseExpr(Function<A,C> f,
                        Function<B,C> g) {
    return g.apply(y);
  }
}
```

Sums in Java

```
public class ErrInt extends Sum<String,Integer> {
  public ErrInt err = new Left("error");
  public ErrInt one = new Right(1);
  public ErrInt add(ErrInt that) {
    return this.caseExpr(
                e -> new Left(e),
                m -> that.caseExpr(
                       e -> new Left(e),
                       n -> new Right(m+n)
                     )
              );
  public ErrInt test = one.add(err);
}
```

Sums in Haskell

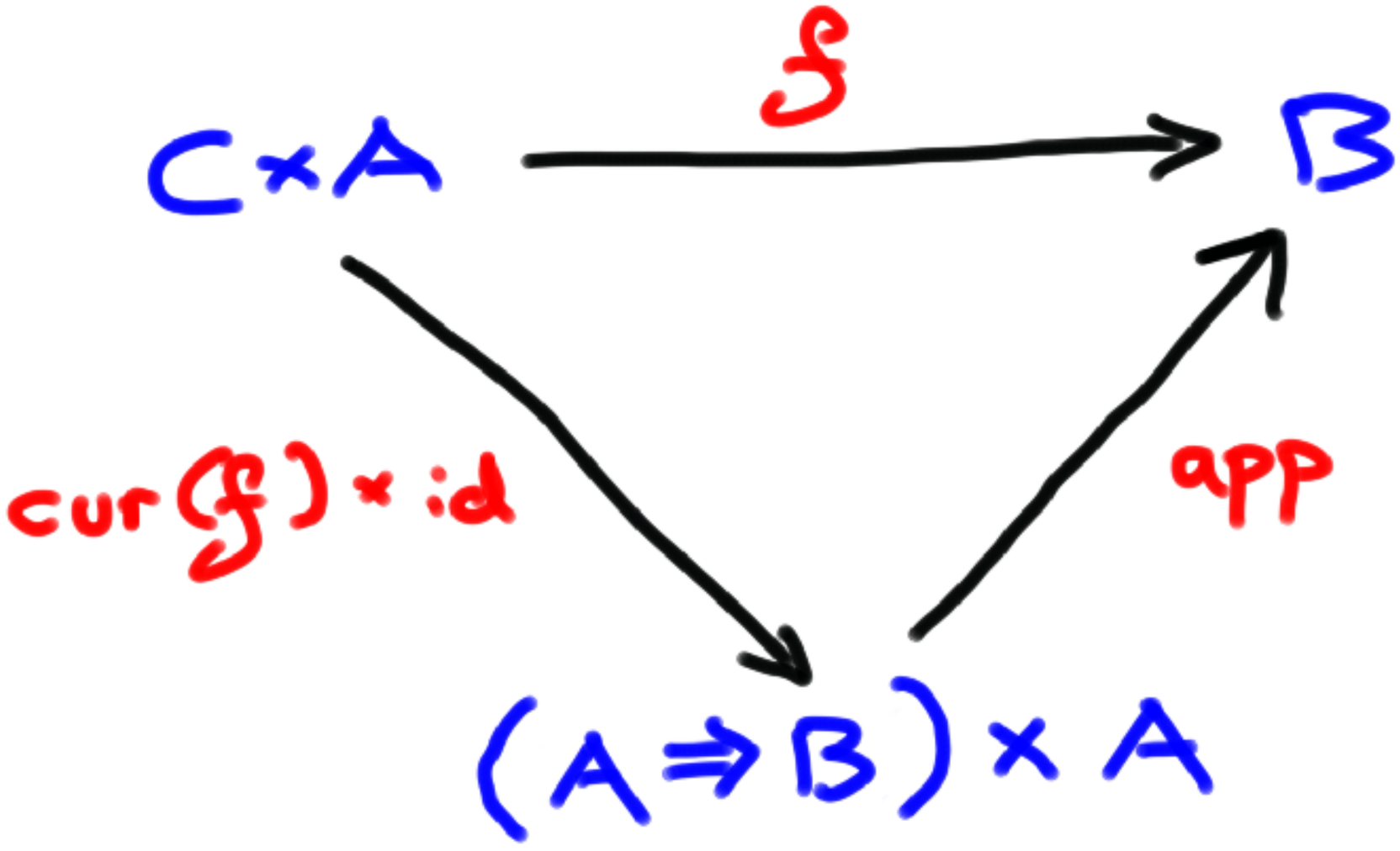
```
data Sum a b = Left a | Right b
```

Sums in Haskell

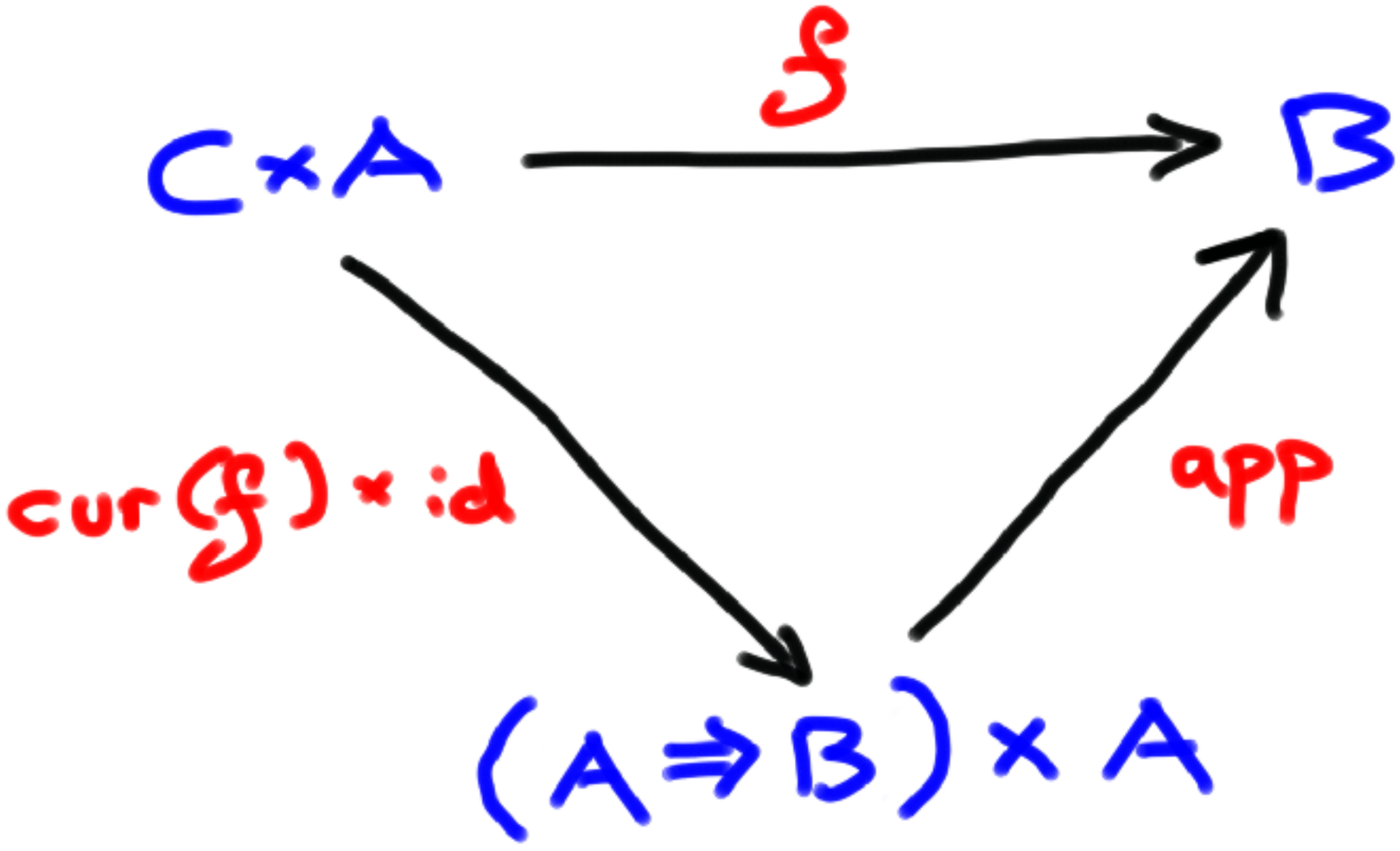
```
type ErrInt = Sum String Int

err = Left “error”
one = Right 1

add :: ErrInt -> ErrInt -> ErrInt
add (Left e)  that         = Left e
add this      (Left e)     = Left e
add (Right m) (Right n) = Right (m+n)

test = add one err
```

Exponentials

EXPONENTIALS



EXPONENTIALS



$$(\mathrm{cur}(f) \times : d)\,;\, \mathrm{app} = f$$

EXPONENTIALS

$$2 \Rightarrow 3 = 3^2$$

$0 \mapsto$ 'a'
$1 \mapsto$ 'a'

$0 \mapsto$ 'a'
$1 \mapsto$ 'b'

$0 \mapsto$ 'a'
$1 \mapsto$ 'c'

$0 \mapsto$ 'b'
$1 \mapsto$ 'a'

$0 \mapsto$ 'b'
$1 \mapsto$ 'b'

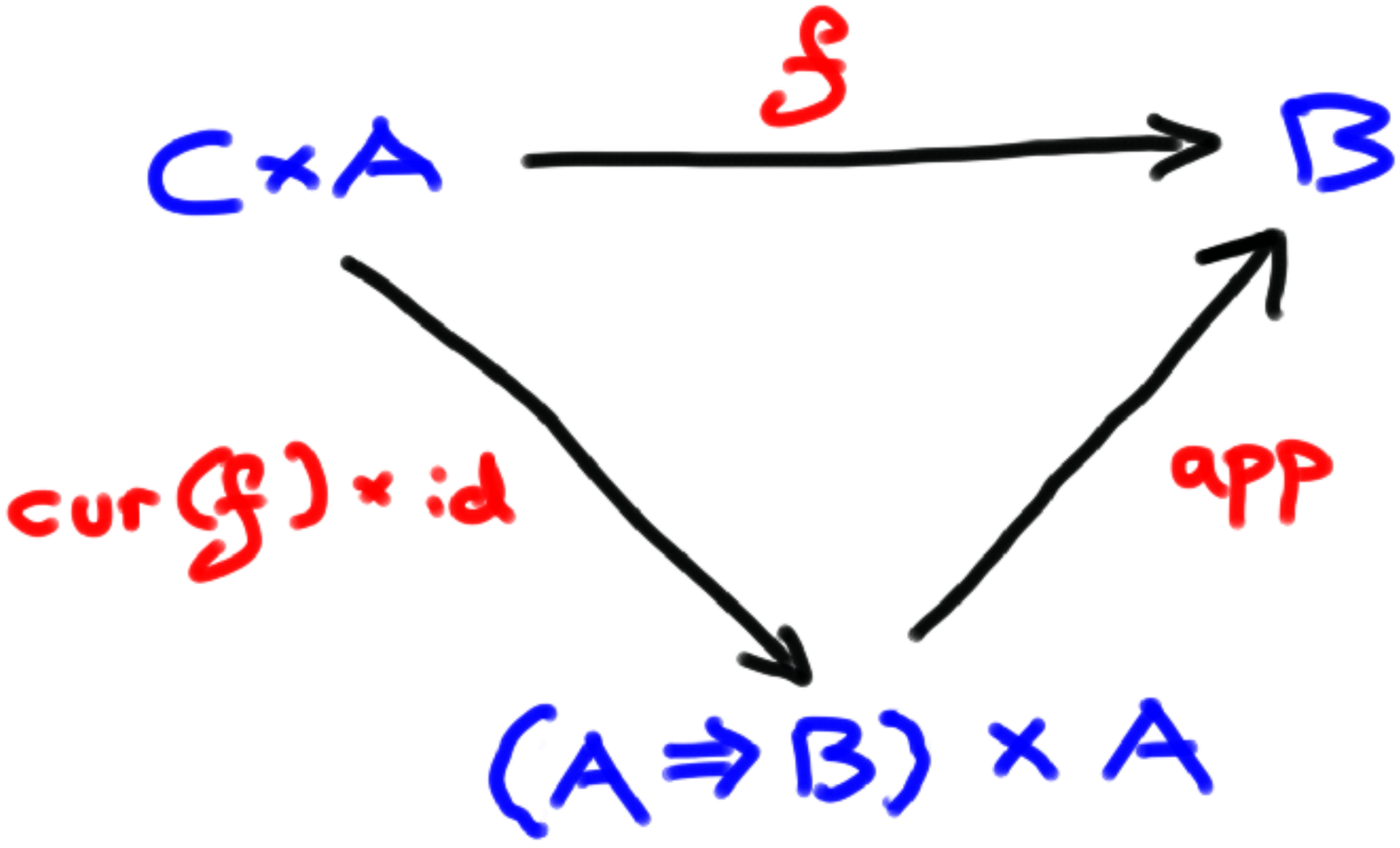
$0 \mapsto$ 'b'
$1 \mapsto$ 'c'

$0 \mapsto$ 'c'
$1 \mapsto$ 'a'

$0 \mapsto$ 'c'
$1 \mapsto$ 'b'

$0 \mapsto$ 'c'
$1 \mapsto$ 'c'

EXPONENTIALS



$$\mathcal{C}(C, A \Rightarrow B) \cong \mathcal{C}(C \times A, B)$$

Exponentials in Java

```
public class Test {
  public Function<Integer,Integer>
    add (Integer n) {
      return x -> x + n;
    }
  public Function<Integer,Integer> incr =
    add(1);
  public Integer three = incr.apply(2);
}
```

Exponentials in Haskell

```
add :: Int -> (Int -> Int)
add n  =  \x -> n + x

incr :: Int -> Int
incr  =  add 1

three :: Int
three =  incr 2
```

Exponentials in Haskell

```
add :: Int -> Int -> Int
add n x  =  n + x

incr :: Int -> Int
incr  =  add 1

three :: Int
three =  incr 2
```

Simplicity & Michelson

Simplicity - Types

$$\frac{}{\mathsf{iden} : A \vdash A} \qquad \frac{s : A \vdash B \qquad t : B \vdash C}{\mathsf{comp}\, s\, t : A \vdash C}$$

$$\frac{}{\mathsf{unit} : A \vdash \mathbb{1}}$$

$$\frac{t : A \vdash B}{\mathsf{injl}\, t : A \vdash B + C} \qquad \frac{t : A \vdash C}{\mathsf{injr}\, t : A \vdash B + C}$$

$$\frac{s : A \times C \vdash D \qquad t : B \times C \vdash D}{\mathsf{case}\, s\, t : (A + B) \times C \vdash D} \qquad \frac{s : A \vdash B \qquad t : A \vdash C}{\mathsf{pair}\, s\, t : A \vdash B \times C}$$

$$\frac{t : A \vdash C}{\mathsf{take}\, t : A \times B \vdash C} \qquad \frac{t : B \vdash C}{\mathsf{drop}\, t : A \times B \vdash C}$$

Figure 1: Typing rules for the terms of core Simplicity.

Simplicity - Semantics

$$
\begin{aligned}
[\![\mathsf{iden}]\!](a) &:= a \\
[\![\mathsf{comp}\, s\, t]\!](a) &:= [\![t]\!]([\![s]\!](a)) \\
[\![\mathsf{unit}]\!](a) &:= \langle\rangle \\
[\![\mathsf{injl}\, t]\!](a) &:= \sigma^{\mathbf{L}}([\![t]\!](a)) \\
[\![\mathsf{injr}\, t]\!](a) &:= \sigma^{\mathbf{R}}([\![t]\!](a)) \\
[\![\mathsf{case}\, s\, t]\!]\langle\sigma^{\mathbf{L}}(a), c\rangle &:= [\![s]\!]\langle a, c\rangle \\
[\![\mathsf{case}\, s\, t]\!]\langle\sigma^{\mathbf{R}}(b), c\rangle &:= [\![t]\!]\langle b, c\rangle \\
[\![\mathsf{pair}\, s\, t]\!](a) &:= \langle [\![s]\!](a), [\![t]\!](a)\rangle \\
[\![\mathsf{take}\, t]\!]\langle a, b\rangle &:= [\![t]\!](a) \\
[\![\mathsf{drop}\, t]\!]\langle a, b\rangle &:= [\![t]\!](b)
\end{aligned}
$$

Michelson - Sums

Operations on unions

- `LEFT 'b`: Pack a value in a union (left case). `:: 'a : 'S -> or 'a 'b : 'S`

```
> LEFT ; C / v :: S   =>   C / (Left v) :: S
```

- `RIGHT 'a`: Pack a value in a union (right case). `:: 'b : 'S -> or 'a 'b : 'S`

```
> RIGHT ; C / v :: S   =>   C / (Right v) :: S
```

- `IF_LEFT bt bf`: Inspect an optional value. `:: or 'a 'b : 'S -> 'c : 'S`

```
   iff   bt :: [ 'a : 'S -> 'c : 'S]
         bf :: [ 'b : 'S -> 'c : 'S]

> IF_LEFT ; C / (Left a) : S   =>   bt ; C / a : S
> IF_LEFT ; C / (Right b) : S   =>   bf ; C / b : S
```

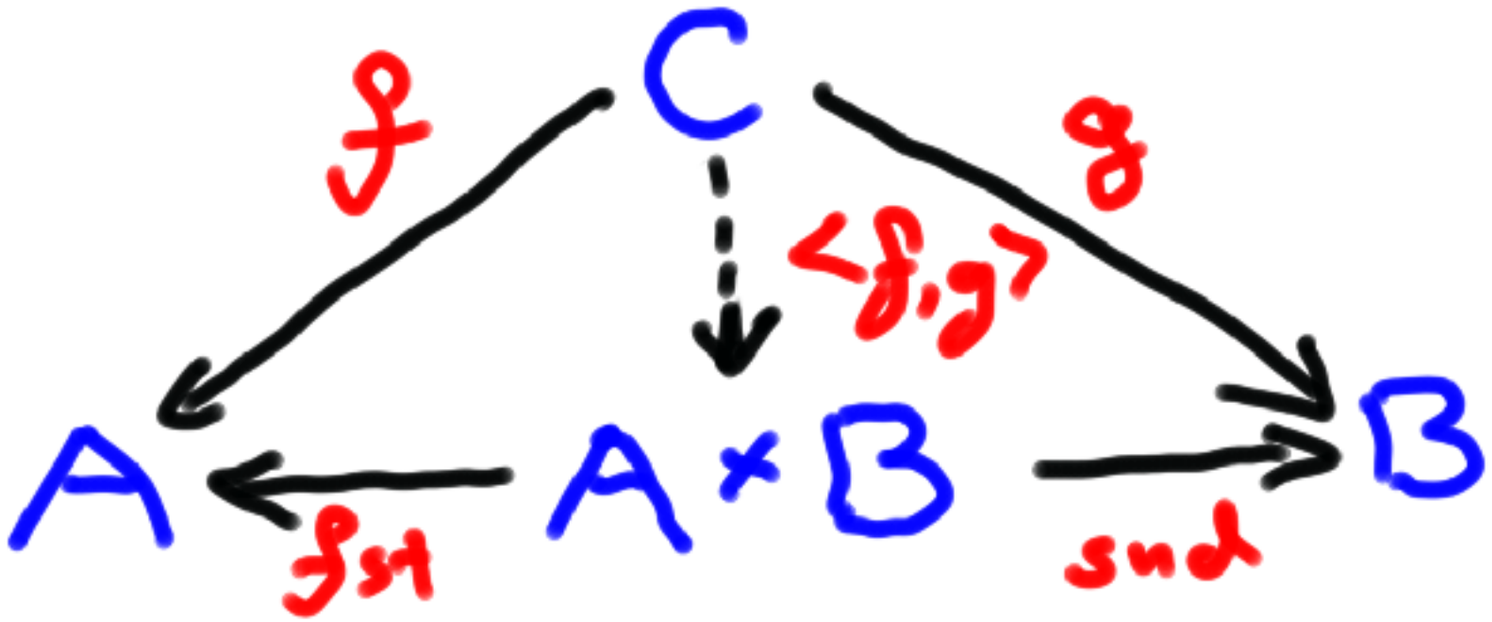
Plutus

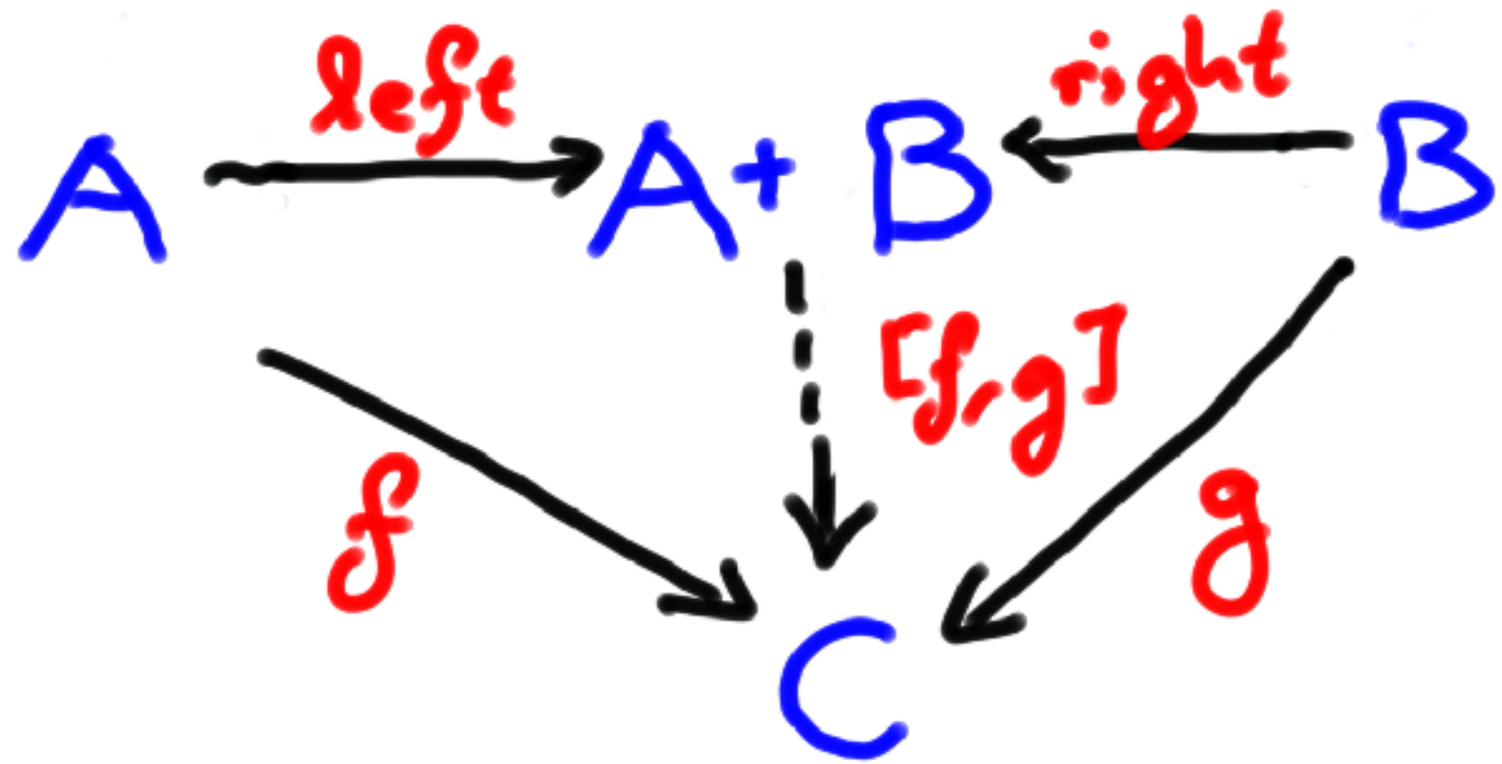
IOHK is hiring 6 programming language engineers

https://iohk.io/careers/#op-235152-functional-compiler-engineer-

Conclusions

DUALS





ISOMORPHISMS

$$\mathcal{C}(C, A \times B) \cong \mathcal{C}(C, A) \times \mathcal{C}(C, B)$$

$$\mathcal{C}(A + B, C) \cong \mathcal{C}(A, C) \times \mathcal{C}(B, C)$$

$$\mathcal{C}(C, A \Rightarrow B) \cong \mathcal{C}(C \times A, B)$$

HIGH SCHOOL

$$(A*B)^C = A^C * B^C$$

$$C^{(A+B)} = C^A * C^B$$

$$(B^A)^C = B^{C*A}$$

Further Reading

- Saunders MacLane, *Categories for the Working Mathematician*

- Benjamin Pierce, *Basic Category Theory for Computer Scientists*

- Bartosz Milewski, *Programming Cafe* (blog)