# Reasoning with strict symmetric monoidal categories in Agda

Malin Altenmüller

Women in EPN, 10th June 2025

THE UNIVERSITY of EDINBURGH
informatics

# Monoidal Categories

Describe structures that can be composed in sequence and in parallel.

## Definition

A category $\mathbf{C}$ is *monoidal* if it is equipped with a functor $\otimes : \mathbf{C} \times \mathbf{C}$, and morphisms:

- left unit $\lambda_A : 1 \otimes A \to A$
- right unit $\rho_A : A \otimes 1 \to A$
- associativity $\alpha_{A,B,C} : (A \otimes B) \otimes C \to A \otimes (B \otimes C)$

satisfying the triangle and pentagon equalities.

# Symmetric Monoidal Categories (SMCs)

## Definition

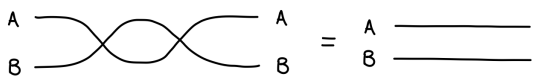
A monoidal category is *symmetric* if it is equipped with a swap operation $\sigma_{A,B} : A \otimes B \to B \otimes A$, such that $\sigma_{A,B} \fatsemi \sigma_{B,A} = 1_{A \otimes B}$ and the hexagon equality holds.

# Symmetric Monoidal Categories (SMCs)

## Definition

A monoidal category is *symmetric* if it is equipped with a swap operation $\sigma_{A,B} : A \otimes B \to B \otimes A$, such that $\sigma_{A,B} \fatsemi \sigma_{B,A} = 1_{A \otimes B}$ and the hexagon equality holds.

Double swap:

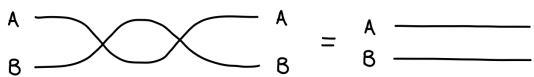


# Symmetric Monoidal Categories (SMCs)

## Definition

A monoidal category is *symmetric* if it is equipped with a swap operation $\sigma_{A,B} : A \otimes B \to B \otimes A$, such that $\sigma_{A,B} \fatsemi \sigma_{B,A} = 1_{A \otimes B}$ and the hexagon equality holds.

Double swap:




Hexagon equality:

$$\begin{array}{ccccc} (A \otimes B) \otimes C & \xrightarrow{\alpha_{A,B,C}} & A \otimes B \otimes C & \xrightarrow{\sigma_{A,B \otimes C}} & (B \otimes C) \otimes A \\ \downarrow{\scriptstyle \sigma_{A,B} \otimes 1_C} & & & & \uparrow{\scriptstyle \alpha_{B,C,A}} \\ (B \otimes A) \otimes C & \xrightarrow{\alpha_{B,A,C}} & B \otimes A \otimes C & \xrightarrow{1_B \otimes \sigma_{A,C}} & B \otimes C \otimes A \end{array}$$

# SMCs in Agda

A category in which all morphisms are invertible:

```
data Ob : Set where
  one : Ob
  _⊗_ : Ob → Ob → Ob
  var : ℕ → Ob
```

# SMCs in Agda

A category in which all morphisms are invertible:

```
data Ob : Set where
  one : Ob
  _⊗_ : Ob → Ob → Ob
  var : ℕ → Ob
```

```
data _↔_ : Ob → Ob → Set where
  id : (A : Ob) → A ↔ A
  _⨟_ : A ↔ B → B ↔ C → A ↔ C
  _⊗_ : A ↔ B → C ↔ D → (A ⊗ C) ↔ (B ⊗ D)
  sym : A ↔ B → B ↔ A
  swap⊗ : (A B : Ob) → A ⊗ B ↔ B ⊗ A
```

# Reasoning with morphisms

The 2-level structure defines equations between morphisms.

```
data _⇔_ : (A ↔ B) → (A ↔ B) → Set where
  id : {c : A ↔ B} → c ⇔ c
  sym : {c d : A ↔ B} → c ⇔ d → d ⇔ c
  _⨾_ : {c d e : A ↔ B} → c ⇔ d → d ⇔ e → c ⇔ e
```

(+ constructors for ⨾ and ⊗ of terms)

# Reasoning with morphisms

The 2-level structure defines equations between morphisms.

```
data _⇔_ : (A ↔ B) → (A ↔ B) → Set where
  id : {c : A ↔ B} → c ⇔ c
  sym : {c d : A ↔ B} → c ⇔ d → d ⇔ c
  _⨾_ : {c d e : A ↔ B} → c ⇔ d → d ⇔ e → c ⇔ e
```

(+ constructors for ⨾ and ⊗ of terms)

For example, we can specify that ⊗ is a functor:

```
id⊗id : (id A ⊗ id B) ⇔ id (A ⊗ B)
```

# Reasoning with morphisms

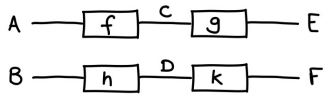
The 2-level structure defines equations between morphisms.

$$
\begin{aligned}
&\text{data } \_\Leftrightarrow\_ : (A \leftrightarrow B) \to (A \leftrightarrow B) \to \text{Set where}\\
&\quad \text{id} : \{c : A \leftrightarrow B\} \to c \Leftrightarrow c\\
&\quad \text{sym} : \{c\ d : A \leftrightarrow B\} \to c \Leftrightarrow d \to d \Leftrightarrow c\\
&\quad \_\fatsemi\_ : \{c\ d\ e : A \leftrightarrow B\} \to c \Leftrightarrow d \to d \Leftrightarrow e \to c \Leftrightarrow e
\end{aligned}
$$

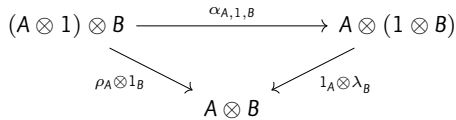
(+ constructors for $\fatsemi$ and $\otimes$ of terms)

For example, we can specify that $\otimes$ is a functor:

$$
\begin{aligned}
&\text{id}\otimes\text{id} : (\text{id}\ A \otimes \text{id}\ B) \Leftrightarrow \text{id}\ (A \otimes B)\\
&\\
&\text{hom}\otimes : \{f : A \leftrightarrow C\}\{g : C \leftrightarrow E\}\{h : B \leftrightarrow D\}\{k : D \leftrightarrow F\}\\
&\qquad \to (f \fatsemi g) \otimes (h \fatsemi k) \Leftrightarrow (f \otimes h) \fatsemi (g \otimes k)
\end{aligned}
$$

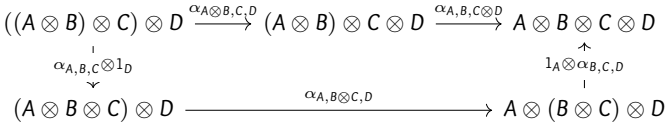


# Triangle equality




triangle : unit⊗r $A$ ⊗ id $B$ ⇔ assoc⊗ {$A$}{one}{$B$} ⨟ (id $A$ ⊗ unit⊗l $B$)

# Pentagon equality




pentagon : assoc⊗ {A ⊗ B} {C} {D} ⨟ assoc⊗ {A} {B} {C ⊗ D}
⇔ assoc⊗ {A} {B} {C} ⊗ id D ⨟ assoc⊗ {A} {B ⊗ C} {D} ⨟ id A ⊗ assoc⊗ {B} {C} {D}

# Coherence isomorphisms on morphisms

To express associativity and unit laws on morphisms, we have to include explicit equations on objects to fix up the types:

$$\text{assoc}\otimes\text{m'} : \{f : A \leftrightarrow B\}\{g : B \leftrightarrow C\}\{h : C \leftrightarrow D\} \to (f \otimes g) \otimes h \Leftrightarrow \text{assoc}\otimes \fatsemi f \otimes g \otimes h \fatsemi \text{sym assoc}\otimes$$

# Coherence isomorphisms on morphisms

To express associativity and unit laws on morphisms, we have to include explicit equations on objects to fix up the types:

$$\mathsf{assoc}\otimes\mathsf{m'} : \{f : A \leftrightarrow B\}\{g : B \leftrightarrow C\}\{h : C \leftrightarrow D\} \to (f \otimes g) \otimes h \Leftrightarrow \mathsf{assoc}\otimes \fatsemi f \otimes g \otimes h \fatsemi \mathsf{sym}\ \mathsf{assoc}\otimes$$

$$\mathsf{unit}\otimes\mathsf{ml'} : \{f : A \leftrightarrow B\} \to \mathsf{id}\ \mathsf{one} \otimes f \Leftrightarrow \mathsf{unit}\otimes\mathsf{l}\ A \fatsemi f \fatsemi \mathsf{sym}\ (\mathsf{unit}\otimes\mathsf{l}\ B)$$

$$\mathsf{unit}\otimes\mathsf{mr'} : \{f : A \leftrightarrow B\} \to f \otimes \mathsf{id}\ \mathsf{one} \Leftrightarrow \mathsf{unit}\otimes\mathsf{r}\ A \fatsemi f \fatsemi \mathsf{sym}\ (\mathsf{unit}\otimes\mathsf{r}\ B)$$

# Coherence isomorphisms on morphisms

To express associativity and unit laws on morphisms, we have to include explicit equations on objects to fix up the types:

$$\mathsf{assoc}\otimes\mathsf{m'} : \{f : A \leftrightarrow B\}\{g : B \leftrightarrow C\}\{h : C \leftrightarrow D\} \to (f \otimes g) \otimes h \Leftrightarrow \mathsf{assoc}\otimes \fatsemi f \otimes g \otimes h \fatsemi \mathsf{sym}\ \mathsf{assoc}\otimes$$

$$\mathsf{unit}\otimes\mathsf{ml'} : \{f : A \leftrightarrow B\} \to \mathsf{id}\ \mathsf{one} \otimes f \Leftrightarrow \mathsf{unit}\otimes\mathsf{l}\ A \fatsemi f \fatsemi \mathsf{sym}\ (\mathsf{unit}\otimes\mathsf{l}\ B)$$

$$\mathsf{unit}\otimes\mathsf{mr'} : \{f : A \leftrightarrow B\} \to f \otimes \mathsf{id}\ \mathsf{one} \Leftrightarrow \mathsf{unit}\otimes\mathsf{r}\ A \fatsemi f \fatsemi \mathsf{sym}\ (\mathsf{unit}\otimes\mathsf{r}\ B)$$

For reasoning with weak MCs, all of the structural equivalences have to be explicit.

# Strict SMCs

**Definition**

In a strict SMC, associativity and unit morphisms are the identity.

# Strict SMCs

**Definition**

In a strict SMC, associativity and unit morphisms are the identity.

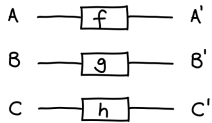
**Lemma**

*In a strict SMC, triangle and pentagon equalities are the identity.*

*Remark*: both paths are the identity, but also the filling of the commutative diagrams.

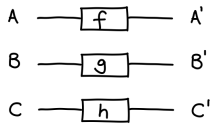
# String Diagrams

In string diagrams the strictness property is trivially satisfied. For example, $(f \otimes g) \otimes h = f \otimes g \otimes h$:



# String Diagrams

In string diagrams the strictness property is trivially satisfied. For example, $(f \otimes g) \otimes h = f \otimes g \otimes h$:




String diagrams depict equivalence classes of morphisms of monoidal categories.

# Reasoning with strict SMCs in Agda

- only computational content is in the swap operations (morphisms are permutations)
- ideally we would only talk about swaps in proofs
- implicit structural equalities have to be explicit in Agda

$$\mathsf{assoc}\otimes\mathsf{m'} : \{f : A \leftrightarrow B\}\{g : B \leftrightarrow C\}\{h : C \leftrightarrow D\} \to (f \otimes g) \otimes h \Leftrightarrow \mathsf{assoc}\otimes \fatsemi f \otimes g \otimes h \fatsemi \mathsf{sym}\ \mathsf{assoc}\otimes$$

# Agda's Rewrite Rules[1]

Adding user-specified definitional equalities to the theory.

[1] Cockx, "Type Theory Unchained: Extending Agda with User-Defined Rewrite Rules".

# Agda's Rewrite Rules[1]

Adding user-specified definitional equalities to the theory.

```
data ℕ : Set where
  zero : ℕ
  suc : ℕ → ℕ
```

```
_+_ : ℕ → ℕ → ℕ
zero + n = n
(suc m) + n = suc (m + n)
```

[1] Cockx, "Type Theory Unchained: Extending Agda with User-Defined Rewrite Rules".

# Agda's Rewrite Rules[1]

Adding user-specified definitional equalities to the theory.

```
data ℕ : Set where
  zero : ℕ
  suc : ℕ → ℕ
```

```
_+_ : ℕ → ℕ → ℕ
zero + n = n
(suc m) + n = suc (m + n)
```

```
unit+ : (a : ℕ) → a + zero ≡ a
unit+ zero = refl
unit+ (suc a) = cong suc (unit+ a)
```

[1] Cockx, "Type Theory Unchained: Extending Agda with User-Defined Rewrite Rules".

# Agda's Rewrite Rules[1]

Adding user-specified definitional equalities to the theory.

```
data ℕ : Set where
  zero : ℕ
  suc : ℕ → ℕ
```

```
_+_ : ℕ → ℕ → ℕ
zero + n = n
(suc m) + n = suc (m + n)
```

```
unit+ : (a : ℕ) → a + zero ≡ a
unit+ zero = refl
unit+ (suc a) = cong suc (unit+ a)
```

```
{-# REWRITE unit+ #-}
```

[1] Cockx, "Type Theory Unchained: Extending Agda with User-Defined Rewrite Rules".

# Agda's Rewrite Rules[1]

Adding user-specified definitional equalities to the theory.

```
data ℕ : Set where
  zero : ℕ
  suc : ℕ → ℕ
```

```
_+_ : ℕ → ℕ → ℕ
zero + n = n
(suc m) + n = suc (m + n)
```

```
unit+ : (a : ℕ) → a + zero ≡ a
unit+ zero = refl
unit+ (suc a) = cong suc (unit+ a)
```

```
assoc+ : (a b c : ℕ) → (a + b) + c ≡ a + b + c
assoc+ zero b c = refl
assoc+ (suc a) b c = cong suc (assoc+ a b c)
```

```
{-# REWRITE unit+ #-}
```

[1] Cockx, "Type Theory Unchained: Extending Agda with User-Defined Rewrite Rules".

# Agda's Rewrite Rules[1]

Adding user-specified definitional equalities to the theory.

```
data ℕ : Set where
  zero : ℕ
  suc : ℕ → ℕ
```

```
_+_ : ℕ → ℕ → ℕ
zero + n = n
(suc m) + n = suc (m + n)
```

```
unit+ : (a : ℕ) → a + zero ≡ a
unit+ zero = refl
unit+ (suc a) = cong suc (unit+ a)
```

```
assoc+ : (a b c : ℕ) → (a + b) + c ≡ a + b + c
assoc+ zero b c = refl
assoc+ (suc a) b c = cong suc (assoc+ a b c)
```

```
{-# REWRITE unit+ #-}
```

```
{-# REWRITE assoc+ #-}
```

[1] Cockx, "Type Theory Unchained: Extending Agda with User-Defined Rewrite Rules".

# Rewrite Example

```
data Vec (A : Set) : ℕ → Set where
  [] : Vec A zero
  _::_ : ∀{n} → (x : A) (xs : Vec A n) → Vec A (suc n)
```

```
_++_ : {A : Set}{m n : ℕ} →
       Vec A m → Vec A n → Vec A (m + n)
[] ++ vs' = vs'
(v :: vs) ++ vs' = v :: (vs ++ vs')
```

# Rewrite Example

```
data Vec (A : Set) : ℕ → Set where
  [] : Vec A zero
  _::_ : ∀{n} → (x : A) (xs : Vec A n) → Vec A (suc n)
```

```
_++_ : {A : Set}{m n : ℕ} →
       Vec A m → Vec A n → Vec A (m + n)
[] ++ vs' = vs'
(v :: vs) ++ vs' = v :: (vs ++ vs')
```

```
assoc++ : {X : Set} → {k l m : ℕ} (as : Vec X k)(bs : Vec X l)(cs : Vec X m) →
          (as ++ bs) ++ cs ≡ as ++ (bs ++ cs)
```

# Rewrite Example

```
data Vec (A : Set) : ℕ → Set where
  [] : Vec A zero
  _::_ : ∀{n} → (x : A) (xs : Vec A n) → Vec A (suc n)
```

```
_++_ : {A : Set}{m n : ℕ} →
       Vec A m → Vec A n → Vec A (m + n)
[] ++ vs' = vs'
(v :: vs) ++ vs' = v :: (vs ++ vs')
```

```
assoc++ : {X : Set} → {k l m : ℕ} (as : Vec X k)(bs : Vec X l)(cs : Vec X m) →
          (as ++ bs) ++ cs ≡ as ++ (bs ++ cs)
```

It typechecks! Even though:

- $(as \mathbin{+\!\!+} bs) \mathbin{+\!\!+} cs : \mathsf{Vec}\ X\ ((k \mathbin{+\!\!+} l) \mathbin{+\!\!+} m)$
- $as \mathbin{+\!\!+} (bs \mathbin{+\!\!+} cs) : \mathsf{Vec}\ X\ (k \mathbin{+\!\!+} (l \mathbin{+\!\!+} m))$

# Use rewrite rules on arbitrary relation

I can use rewrite rules on relations that I have specified myself.

- idea: work with equivalence classes of the relation
- choose one representative and rewrite everything else to it (e.g. associate to the right)

# Use rewrite rules on arbitrary relation

I can use rewrite rules on relations that I have specified myself.

- idea: work with equivalence classes of the relation
- choose one representative and rewrite everything else to it (e.g. associate to the right)

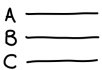
Plan: use it to declare coherence isomorphisms of SMCs as definitional equalities.

- extract computationally relevant part of a proof
- proof in Agda to look like the paper one

# Rewrite equations on objects

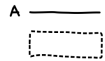
$\text{assoc}\otimes : (A \otimes B) \otimes C \leftrightarrow A \otimes B \otimes C$

{-# REWRITE assoc⊗ #-}



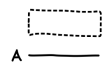
# Rewrite equations on objects

$\mathsf{unit}\otimes\mathsf{l} : (A : \mathsf{Ob}) \to (\mathsf{one} \otimes A) \leftrightarrow A$

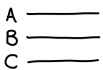



$\mathsf{assoc}\otimes : (A \otimes B) \otimes C \leftrightarrow A \otimes B \otimes C$

$\mathsf{unit}\otimes\mathsf{r} : (A : \mathsf{Ob}) \to (A \otimes \mathsf{one}) \leftrightarrow A$



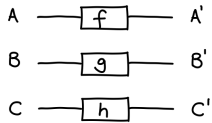

{-# REWRITE assoc⊗ #-}




{-# REWRITE unit⊗l unit⊗r #-}

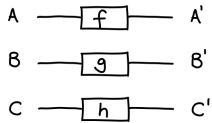
# Equations on morphisms

$$\text{assoc}\otimes\text{mr} : \{f : A \leftrightarrow B\}\{g : B \leftrightarrow C\}\{h : C \leftrightarrow D\} \to (f \otimes g) \otimes h \Leftrightarrow f \otimes g \otimes h$$



# Equations on morphisms

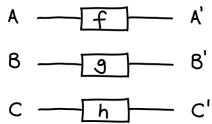
$\text{assoc}\otimes\text{mr} : \{f : A \leftrightarrow B\}\{g : B \leftrightarrow C\}\{h : C \leftrightarrow D\} \to (f \otimes g) \otimes h \Leftrightarrow f \otimes g \otimes h$




$\text{unit}\otimes\text{ml} : \{f : A \leftrightarrow B\} \to \text{id one} \otimes f \Leftrightarrow f$
$\text{unit}\otimes\text{mr} : \{f : A \leftrightarrow B\} \to f \otimes \text{id one} \Leftrightarrow f$

# Equations on morphisms

$$\text{assoc}\otimes\text{mr} : \{f : A \leftrightarrow B\}\{g : B \leftrightarrow C\}\{h : C \leftrightarrow D\} \to (f \otimes g) \otimes h \Leftrightarrow f \otimes g \otimes h$$




$$\text{unit}\otimes\text{ml} : \{f : A \leftrightarrow B\} \to \text{id one} \otimes f \Leftrightarrow f$$
$$\text{unit}\otimes\text{mr} : \{f : A \leftrightarrow B\} \to f \otimes \text{id one} \Leftrightarrow f$$

Add these equations as definitional equalities, too!

```
{-# REWRITE assoc⊗mr unit⊗ml unit⊗mr #-}
```

# Strict SMCs in Agda

Even stronger: we can now strictify the category, by declaring…

```
assoc⊗=id : assoc⊗ {A}{B}{C} ⇔ id (A ⊗ B ⊗ C)
unit⊗l=id : unit⊗l A ⇔ id A
unit⊗r=id : unit⊗r A ⇔ id A
```

# Strict SMCs in Agda

Even stronger: we can now strictify the category, by declaring…

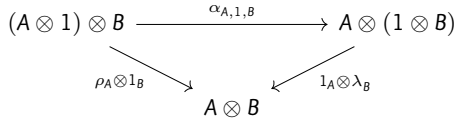
```
assoc⊗=id : assoc⊗ {A}{B}{C} ⇔ id (A ⊗ B ⊗ C)
unit⊗l=id : unit⊗l A ⇔ id A
unit⊗r=id : unit⊗r A ⇔ id A
```

…and immediately rewriting by these equations.

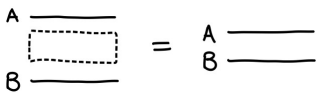
Additionally, we rewrite by functoriality of $\otimes$, e.g.

```
id⊗id : (id t1 ⊗↓ id t2) ⇔ id (t1 ⊗ t2)
```

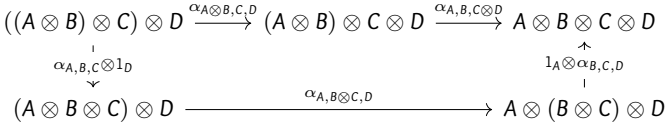
# Triangle Equality in a strict SMC



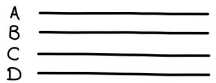

```
triangle : {A B : Ob} → unit⊗r A ⊗ id B ⇔ assoc⊗r {A}{one}{B} ⨟ (id A ⊗ unit⊗l B)
triangle = id
```



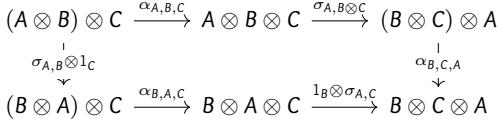
# Pentagon Equality in a strict SMC




```
pentagon : {A B C D : Ob} →
           assoc⊗r {A ⊗ B} {C} {D} ⨟ assoc⊗r {A} {B} {C ⊗ D}
         ⇔ assoc⊗r {A} {B} {C} ⊗ id D ⨟ assoc⊗r {A} {B ⊗ C} {D} ⨟ id A ⊗ assoc⊗r {B} {C} {D}
pentagon = id
```



# Hexagon in a strict SMC



# What to use it for?

I'm interested in rig categories[2]:

- Structural foundation for the semantics of quantum computation.
- Contain two monoidal structures $(\otimes,1)$ and $(\oplus,0)$.
- Distributive law between them: $A \otimes (B \oplus C) = (A \otimes B) \oplus (A \otimes C)$.
- a lot of coherence conditions! Including a lot about structural equivalences.

[2] Heunen and Kaarsgaard, "Quantum information effects".

# Two versions of not[3]

Type of booleans:

$$\mathsf{bool} = \mathsf{one} \oplus \mathsf{one}$$

[3] Carette and Sabry, “Computing with Semirings and Weak Rig Groupoids”.

# Two versions of not[3]

Type of booleans:

```
bool = one ⊕ one
```

Two implementations of the not operation:

```
not1 : bool ↔ bool
not1 = swap⊕ one one
```


[3] Carette and Sabry, “Computing with Semirings and Weak Rig Groupoids”.

# Two versions of not[3]

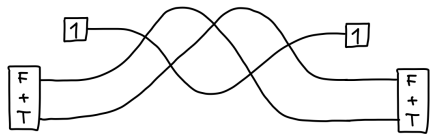
Type of booleans:

```
bool = one ⊕ one
```

Two implementations of the not operation:

```
not1 : bool ↔ bool
not1 = swap⊕ one one
```


```
not2 : bool ↔ bool
not2 = sym (unit⊗l (one ⊕ one))
       ⨟ swap⊗ one (one ⊕ one)
       ⨟ swap⊕ one one ⊗↓ id one
       ⨟ swap⊗ (one ⊕ one) one
       ⨟ unit⊗l (one ⊕ one)
```




[3] Carette and Sabry, “Computing with Semirings and Weak Rig Groupoids”.

# Not is not

```
same-not : not2 ⇔ not1
same-not =
  swap⊗-nat {one ⊕ one}{one ⊕ one}{one}{one}
            {swap⊕ one one}{id one}
            ⨟↓ id {c = swap⊗ (one ⊕ one) one}
  ⨟ (id ⨟↓ swap⊗2 {one}{one ⊕ one}
  ⨟ unit⨟↓r {one ⊗ (one ⊕ one)}
  ⨟ unit⊗↓l (swap⊕ one one))
```

# Not is not

```
same-not : not2 ⇔ not1
same-not =
  swap⊗-nat {one ⊕ one}{one ⊕ one}{one}{one}
            {swap⊕ one one}{id one}
            ⨟↓ id {c = swap⊗ (one ⊕ one) one}
  ⨟ (id ⨟↓ swap⊗2 {one}{one ⊕ one}
  ⨟ unit⨟↓r {one ⊗ (one ⊕ one)}
  ⨟ unit⊗↓l (swap⊕ one one))
```

```
negEx : NOT2 ⇔ NOT1
negEx = uniti⋆l ⊙ (Pi0.swap⋆ ⊙ ((Pi0.swap₊ ⊗ id↔) ⊙ (Pi0.swap⋆ ⊙ unite⋆l)))
    ⇔⟨ id⇔ ⊡ assoc⊙l ⟩
  uniti⋆l ⊙ ((Pi0.swap⋆ ⊙ (Pi0.swap₊ ⊗ id↔)) ⊙ (Pi0.swap⋆ ⊙ unite⋆l))
    ⇔⟨ id⇔ ⊡ (swapl⋆⇔ ⊡ id⇔) ⟩
uniti⋆l ⊙ (((id↔ ⊗ Pi0.swap₊) ⊙ Pi0.swap⋆) ⊙ (Pi0.swap⋆ ⊙ unite⋆l))
  ⇔⟨ id⇔ ⊡ assoc⊙r ⟩
uniti⋆l ⊙ ((id↔ ⊗ Pi0.swap₊) ⊙ (Pi0.swap⋆ ⊙ (Pi0.swap⋆ ⊙ unite⋆l)))
  ⇔⟨ id⇔ ⊡ (id⇔ ⊡ assoc⊙l) ⟩
uniti⋆l ⊙ ((id↔ ⊗ Pi0.swap₊) ⊙ ((Pi0.swap⋆ ⊙ Pi0.swap⋆) ⊙ unite⋆l))
  ⇔⟨ id⇔ ⊡ (id⇔ ⊡ (linv⊙l ⊡ id⇔)) ⟩
uniti⋆l ⊙ ((id↔ ⊗ Pi0.swap₊) ⊙ (id↔ ⊙ unite⋆l))
  ⇔⟨ id⇔ ⊡ (id⇔ ⊡ idl⊙l) ⟩
uniti⋆l ⊙ ((id↔ ⊗ Pi0.swap₊) ⊙ unite⋆l)
  ⇔⟨ assoc⊙l ⟩
(uniti⋆l ⊙ (id↔ ⊗ Pi0.swap₊)) ⊙ unite⋆l
  ⇔⟨ unitil⋆⇔l ⊡ id⇔ ⟩
(Pi0.swap₊ ⊙ uniti⋆l) ⊙ unite⋆l
  ⇔⟨ assoc⊙r ⟩
Pi0.swap₊ ⊙ (uniti⋆l ⊙ unite⋆l)
  ⇔⟨ id⇔ ⊡ linv⊙l ⟩
Pi0.swap₊ ⊙ id↔
  ⇔⟨ idr⊙l ⟩
Pi0.swap₊ □
```

# Confluence checking

- Agda has a local-confluence-check pragma for rewrite rules
- this does not interact well with rewrite rules that typecheck because of other rewrite rules:

$$\mathsf{assoc}\otimes\mathsf{mr} : \{f : A \leftrightarrow B\}\{g : B \leftrightarrow C\}\{h : C \leftrightarrow D\} \to (f \otimes g) \otimes h \Leftrightarrow f \otimes g \otimes h$$

- check confluence by hand…

# Summary

- strict SMC contain a lot of trivial structural coherence isomorphisms
- with Agda's rewrite rules these can be implicit in the formalisation
- can extract the computational interesting part of a proof

# Summary

- strict SMC contain a lot of trivial structural coherence isomorphisms
- with Agda's rewrite rules these can be implicit in the formalisation
- can extract the computational interesting part of a proof

Some future ideas:

- apply to other flavours of MC (e.g. braided)
- explore rewriting of setoid equalities in Agda
- rewriting heterogeneous equalities?

# Summary

- strict SMC contain a lot of trivial structural coherence isomorphisms
- with Agda's rewrite rules these can be implicit in the formalisation
- can extract the computational interesting part of a proof

Some future ideas:

- apply to other flavours of MC (e.g. braided)
- explore rewriting of setoid equalities in Agda
- rewriting heterogeneous equalities?

*Thank you for your attention!*

Thank you for your attention!

## Reasoning with strict symmetric monoidal categories in Agda

Malin Altenmüller
malin.altenmuller@ed.ac.uk

Women in EPN, 10th June 2025

THE UNIVERSITY of EDINBURGH
informatics


Carette, Jacques and Amr Sabry. “Computing with Semirings and Weak Rig Groupoids”. In: *Programming Languages and Systems - 25th European Symposium on Programming, ESOP 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*. Ed. by Peter Thiemann. Vol. 9632. Lecture Notes in Computer Science. Springer, 2016, pp. 123–148. DOI: 10.1007/978-3-662-49498-1_6. URL: https://doi.org/10.1007/978-3-662-49498-1_6.


Cockx, Jesper. “Type Theory Unchained: Extending Agda with User-Defined Rewrite Rules”. In: *25th International Conference on Types for Proofs and Programs, TYPES 2019, June 11-14, 2019, Oslo, Norway*. Ed. by Marc Bezem and Assia Mahboubi. Vol. 175. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019, 2:1–2:27. DOI: 10.4230/LIPICS.TYPES.2019.2. URL: https://doi.org/10.4230/LIPIcs.TYPES.2019.2.


Heunen, Chris and Robin Kaarsgaard. “Quantum information effects”. In: *Proc. ACM Program. Lang.* 6.POPL (2022), pp. 1–27. DOI: 10.1145/3498663. URL: https://doi.org/10.1145/3498663.