The Relation with Types

Marcin Kostrzewa

@mmkostrzewa

“Types are helpful for proving useful properties of programs”

Most Typing Advocates

How many properties have you proven so far?

Teaser problem

Given a function `f` of type

```
f :: forall A . [A] -> [A]
```

what can we say about it?

First intuitions

```
f :: forall A . [A] -> [A]
```

First intuitions

```
f :: forall A . [A] -> [A]

reverse :: forall A . [A] -> [A]

tail :: forall A . [A] -> [A]
```

First intuitions

We can't work directly with elements of type `A`

```
f :: forall A . [A] -> [A]

reverse :: forall A . [A] -> [A]

tail :: forall A . [A] -> [A]
```

Towards useful interpretation of types

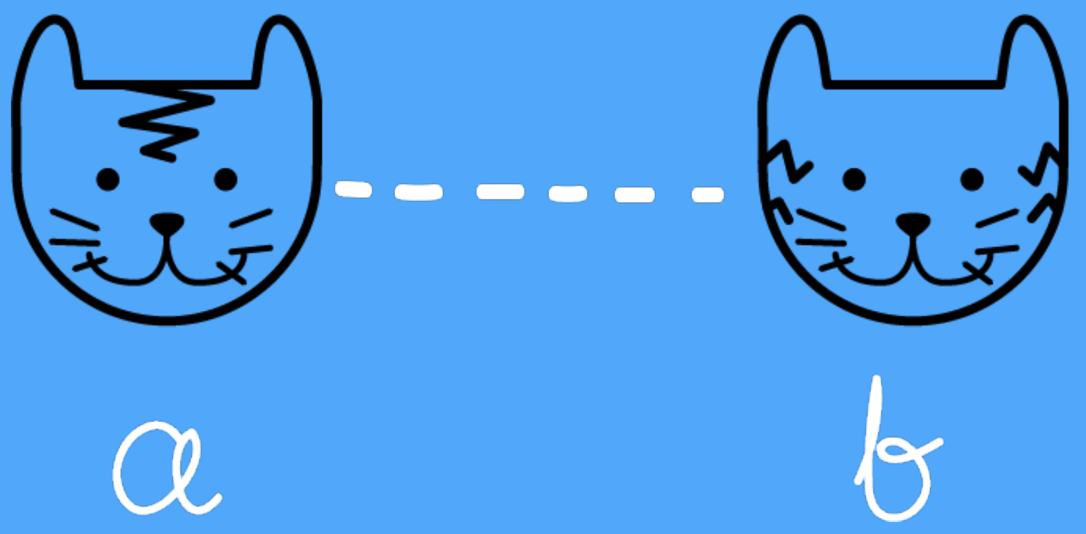
Warning!

Scary stuff may appear.

Relations

A set of pairs of related elements.

$(a, b) \in \mathcal{R}$ means that a and b are related in $\mathcal{R}$.



Relations

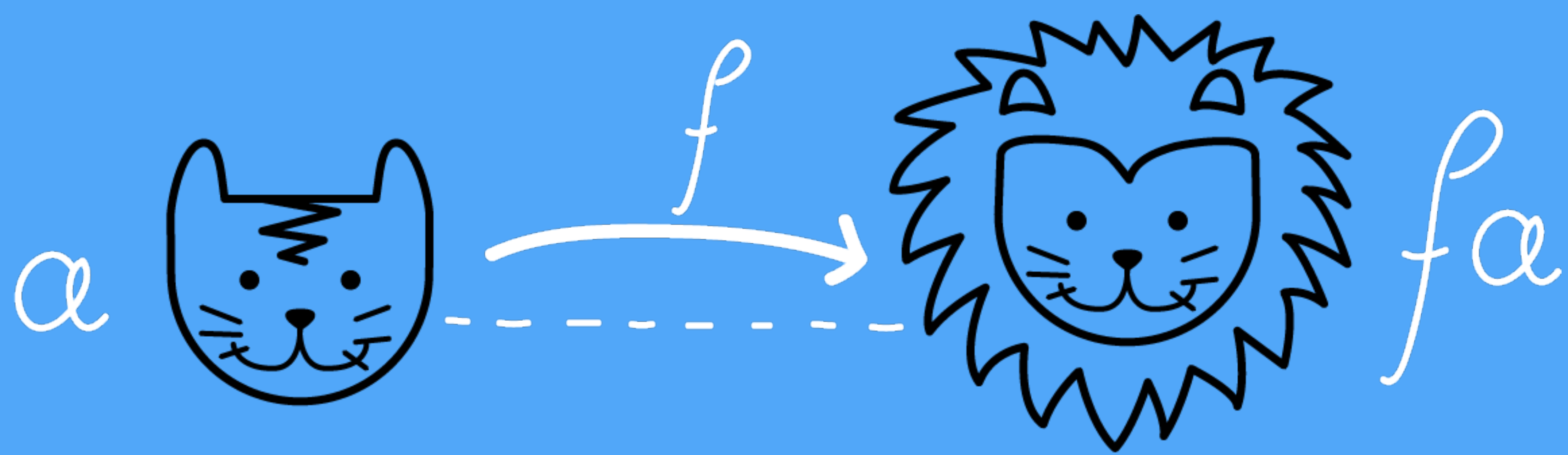
Identity relation

Identity relation on A: $I_A = \{(a,a) \mid a \in A\}$.

Relations

Functions

Function $f : A \to B$ as a relation: $\{(a, fa) \mid a \in A\}$.



Types as relations

Types as relations

$\mathcal{R}$ is a mapping interpreting types as relations.

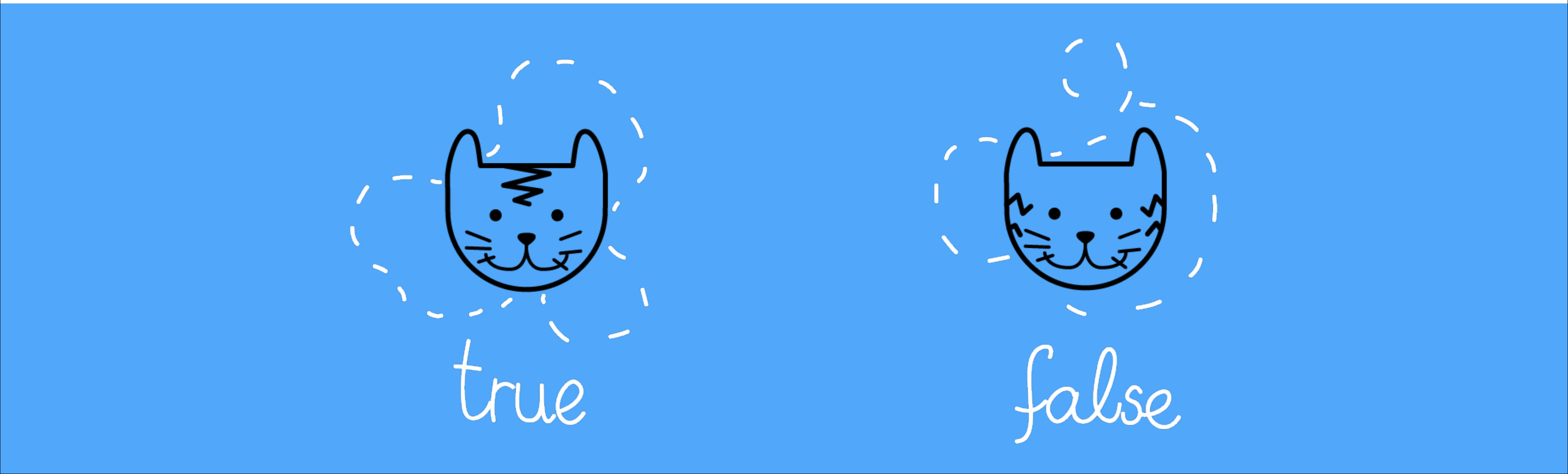
$\mathcal{R}_{\mathcal{A}}$ denotes a relation obtained for $\mathcal{A}$

Types as relations

Primitive types

$$\mathcal{R}_{\texttt{Bool}} = I_{\texttt{Bool}} = \{(\texttt{True}, \texttt{True}), (\texttt{False}, \texttt{False})\}$$

$$\mathcal{R}_{\texttt{Int}} = I_{\texttt{Int}} = \{\ldots, (-1, -1), (0, 0), (1, 1), \ldots\}$$



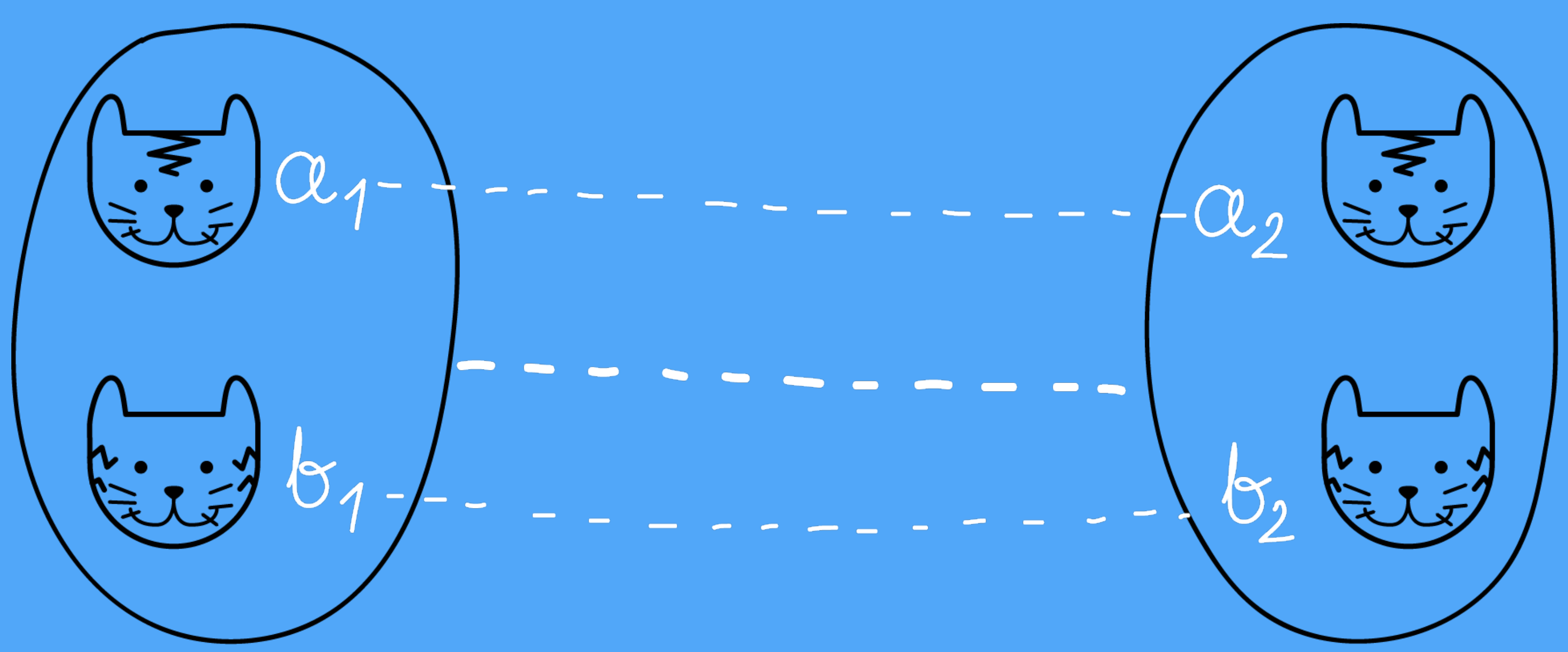
Types as relations

Pairs

$$((a_1, b_1), (a_2, b_2)) \in \mathcal{R}_{\mathcal{A} \times \mathcal{B}}$$

if and only if

$(a_1, a_2) \in \mathcal{A}$ and $(b_1, b_2) \in \mathcal{B}$



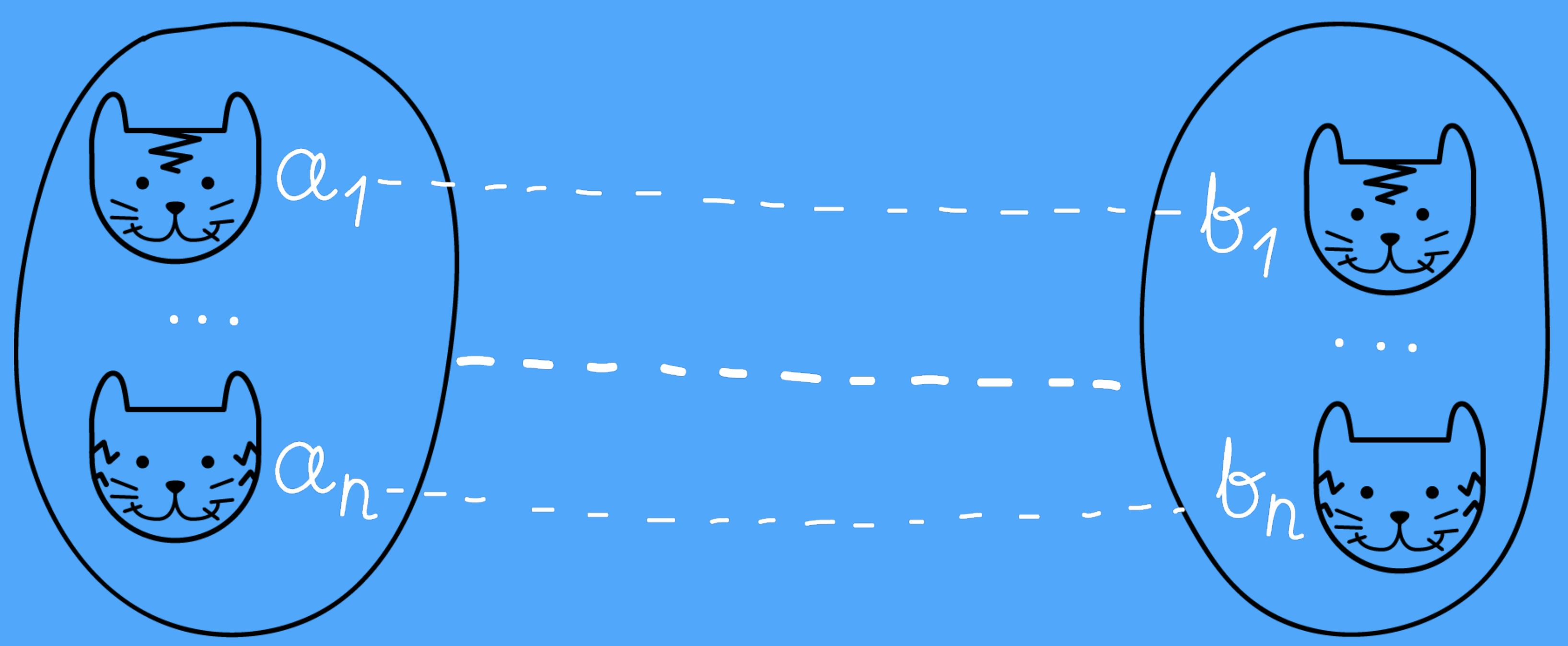
Types as relations

Lists

$$([a_1, a_2, \ldots, a_n], [b_1, b_2, \ldots, b_n]) \in \mathcal{R}_{[\mathcal{A}]}$$

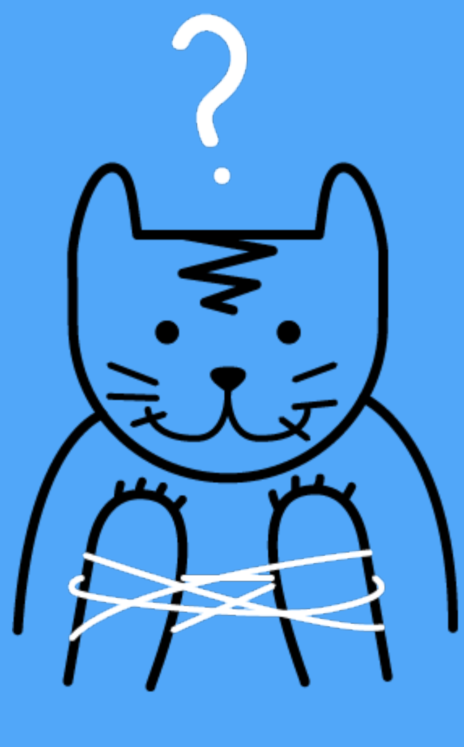
if and only if

$(a_1, b_1) \in \mathcal{A}$ and $(a_2, b_2) \in \mathcal{A}$ … and $(a_n, b_n) \in \mathcal{A}$



Quick test

If f is a function, then what is $\mathcal{R}_{[f]}$?



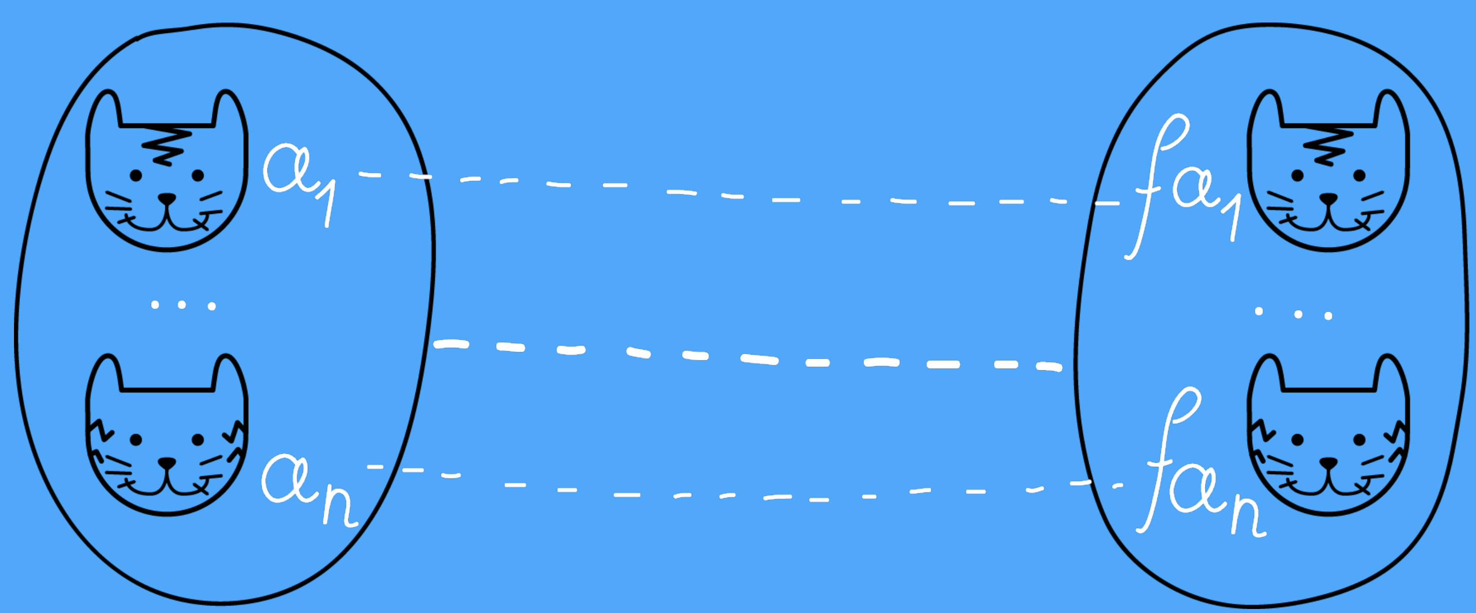
Quick test

If f is a function, then what is $\mathcal{R}_{[f]}$?

a_1 is related to $f\,a_1$

a_2 is related to $f\,a_2$

…



Quick test

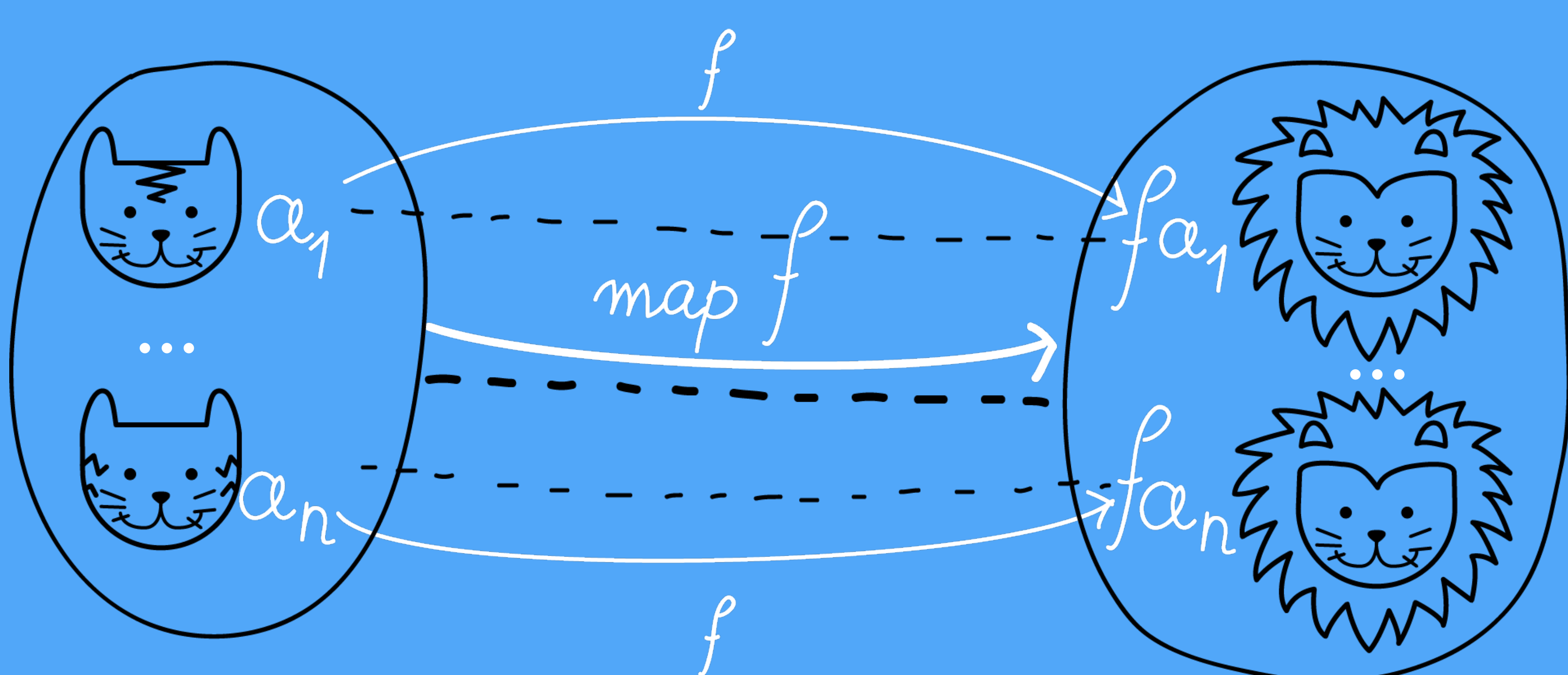
If f is a function, then what is $\mathcal{R}_{[f]}$?

a_1 is related to $f\,a_1$

a_2 is related to $f\,a_2$

…

$[a_1, \ldots, a_n]$ is related to $map \quad f \quad [a_1, \ldots, a_n]$



Types as relations

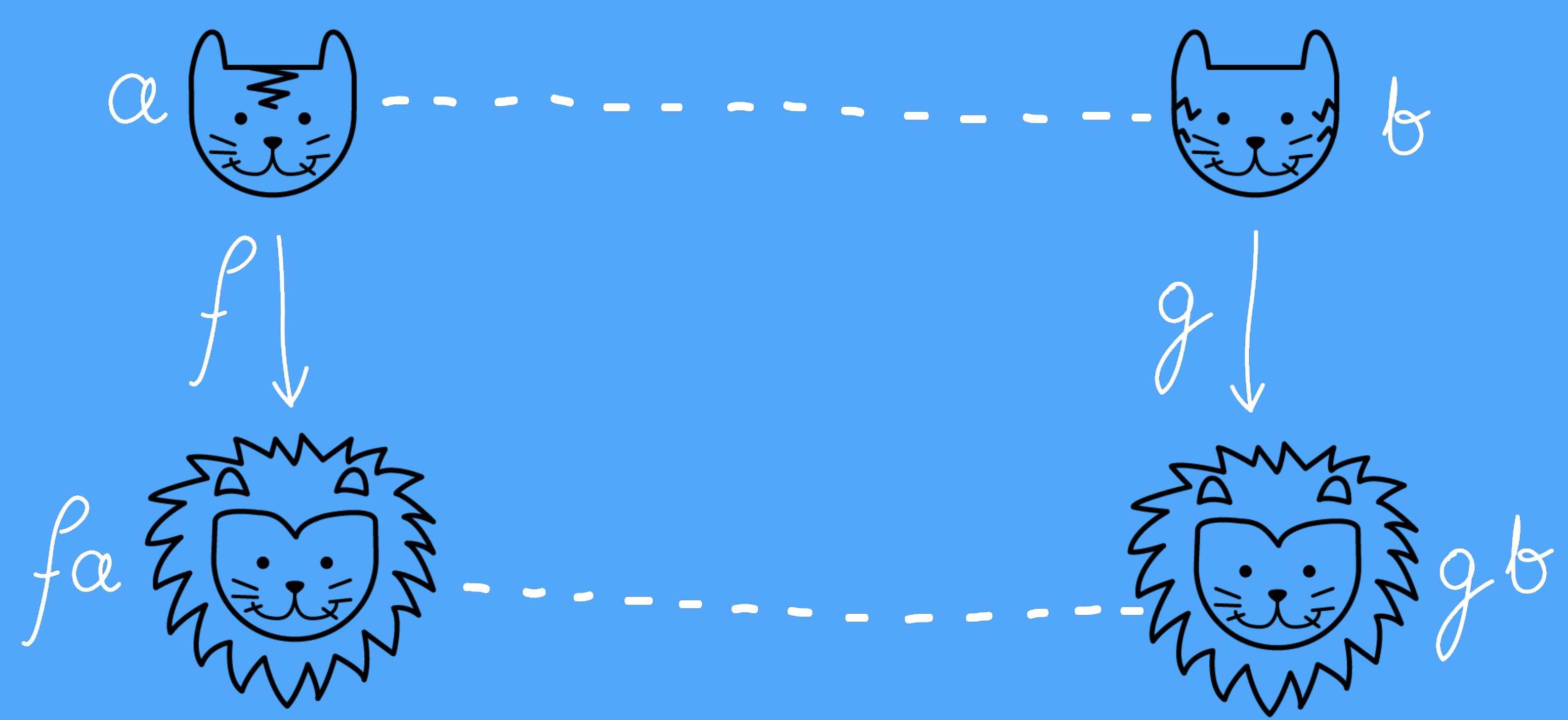
Functions

Map related arguments to related values:

$(f, g) \in \mathcal{R}_{\mathcal{A} \to \mathcal{B}}$

if and only if

$(f\,a, g\,b) \in \mathcal{B}$ for any $(a, b) \in \mathcal{A}$



Types as relations

Polymorphism

Types as relations

Polymorphism

Parametric type is a family of types.

e.g. `forall x . x -> x = { Int -> Int,`
```
                           Bool -> Bool,
                           [Int] -> [Int],
                           ... }
```

Types as relations

Polymorphism

Parametric type is a family of types.

e.g. `forall x . x -> x = { Int -> Int,`
`Bool -> Bool,`
`[Int] -> [Int],`
`... }`

A function belongs to a parametric type if it can be "instantiated" to any type in this family.

e.g. $\texttt{id}_{\texttt{Bool}}$ `:: Bool -> Bool`

$\texttt{id}_{\texttt{Int}}$ `:: Int -> Int`

$\texttt{id}_{\texttt{[Int]}}$ `:: [Int] -> [Int]`

Types as relations

Polymorphism

$(a, b) \in \mathcal{R}_{\forall \mathcal{X}.\mathcal{Y}}$

if and only if

$(a_X, b_X) \in \mathcal{Y}_X$ for any relation X

Putting it together

The Parametricity Theorem

Any term of type $\mathtt{T}$ is related to itself in $\mathcal{R}_{\mathtt{T}}$.

Well, there’s the theory

But how is it supposed to help us?

Warm up exercise #1

The type of identity

```
g :: forall A . A -> A
```

What are the possible implementations?

Warm up exercise #1

The type of identity

```
g :: forall A . A -> A
```

What are the possible implementations?

The parametricity theorem says:

$$(\mathrm{g}, \mathrm{g}) \in \mathcal{R}_{\forall \mathcal{A}. \mathcal{A} \to \mathcal{A}}$$

Warm up exercise #1

The type of identity

```
g :: forall A . A -> A
```

$$(\mathrm{g}, \mathrm{g}) \in \mathcal{R}_{\forall \mathcal{A}.\mathcal{A} \to \mathcal{A}}$$

Warm up exercise #1

The type of identity

```
g :: forall A . A -> A
```

$$(\mathrm{g}, \mathrm{g}) \in \mathcal{R}_{\forall \mathcal{A}.\mathcal{A} \to \mathcal{A}}$$

It promises to work with any relation substituted for $\mathcal{A}$.

Warm up exercise #1

The type of identity

```
g :: forall A . A -> A
```

$$(\mathrm{g}, \mathrm{g}) \in \mathcal{R}_{\forall \mathcal{A}.\mathcal{A} \to \mathcal{A}}$$

It promises to work with any relation substituted for $\mathcal{A}$.

Pick your favorite value, e.g. `True` and the relation

$$\mathcal{A} = \{(\mathtt{True}, \mathtt{True})\}$$

Warm up exercise #1

The type of identity

$$(\mathrm{g}, \mathrm{g}) \in \mathcal{R}_{\mathcal{A} \to \mathcal{A}}$$

where $\mathcal{A} = \{(\mathtt{True}, \mathtt{True})\}$

Warm up exercise #1

The type of identity

$$(\mathrm{g}, \mathrm{g}) \in \mathcal{R}_{\mathcal{A} \to \mathcal{A}}$$

where $\mathcal{A} = \{(\mathtt{True}, \mathtt{True})\}$

It maps related inputs into related outputs.

Warm up exercise #1

The type of identity

$$(\mathrm{g}, \mathrm{g}) \in \mathcal{R}_{\mathcal{A} \to \mathcal{A}}$$

where $\mathcal{A} = \{(\mathtt{True}, \mathtt{True})\}$

It maps related inputs into related outputs.

It maps **`(True, True)`** into **`(True, True)`**!

Warm up exercise #1

The type of identity

$$(\mathtt{g}, \mathtt{g}) \in \mathcal{R}_{\mathcal{A} \to \mathcal{A}}$$

where $\mathcal{A} = \{(\mathtt{True}, \mathtt{True})\}$

It maps related inputs into related outputs.

It maps `(True, True)` into `(True, True)`!

```
g True = True
```

Warm up exercise #1

The type of identity

We could pick anything in place of `True`.

Warm up exercise #1

The type of identity

We could pick anything in place of `True`.

```
g 1 = 1
```

Warm up exercise #1

The type of identity

We could pick anything in place of `True`.

```
g 1 = 1
g (\x -> x) = (\x -> x)
```

Warm up exercise #1

The type of identity

We could pick anything in place of `True`.

```
g 1 = 1
g (\x -> x) = (\x -> x)
…
g = id
```

That was easy, huh?

What else can we do?

Warm up exercise #2

Polymorphic equality

Is it possible to have a function that would meaningfully compare values of any type?

```
eq :: forall A . A -> A -> Bool
```

Warm up exercise #2

Polymorphic equality

```
eq :: forall A . A -> A -> Bool
```

The parametricity theorem yields

$$(\mathtt{eq}, \mathtt{eq}) \in \mathcal{R}_{\forall \mathcal{A}.\mathcal{A} \to \mathcal{A} \to \mathtt{Bool}}$$

Warm up exercise #2

Polymorphic equality

```
eq :: forall A . A -> A -> Bool
```

$$(\mathrm{eq}, \mathrm{eq}) \in \mathcal{R}_{\forall \mathcal{A}.\mathcal{A} \to \mathcal{A} \to \mathtt{Bool}}$$

Warm up exercise #2

Polymorphic equality

```
eq :: forall A . A -> A -> Bool
```

$$(\mathrm{eq}, \mathrm{eq}) \in \mathcal{R}_{\forall \mathcal{A}.\mathcal{A} \to \mathcal{A} \to \mathtt{Bool}}$$

It promises to work with any relation substituted for $\mathcal{A}$.

Warm up exercise #2

Polymorphic equality

```
eq :: forall A . A -> A -> Bool
```

$$(\mathtt{eq}, \mathtt{eq}) \in \mathcal{R}_{\forall \mathcal{A}.\mathcal{A} \to \mathcal{A} \to \mathtt{Bool}}$$

It promises to work with any relation substituted for $\mathcal{A}$.

Let's try a full relation $\mathcal{T}$ — any two elements are related.

Warm up exercise #2

Polymorphic equality

```
eq :: forall A . A -> A -> Bool
```

$$(\mathsf{eq}, \mathsf{eq}) \in \mathcal{R}_{\forall \mathcal{A}.\mathcal{A} \to \mathcal{A} \to \mathtt{Bool}}$$

It promises to work with any relation substituted for $\mathcal{A}$.

Let's try a full relation $\mathcal{T}$ — any two elements are related.

$$(\mathsf{eq}, \mathsf{eq}) \in \mathcal{R}_{\mathcal{T} \to \mathcal{T} \to \mathtt{Bool}}$$

Warm up exercise #2

Polymorphic equality

$$(\mathtt{eq}, \mathtt{eq}) \in \mathcal{R}_{\mathcal{T} \to \mathcal{T} \to \mathtt{Bool}}$$

Warm up exercise #2

Polymorphic equality

$$(\mathtt{eq}, \mathtt{eq}) \in \mathcal{R}_{\mathcal{T} \to \mathcal{T} \to \mathtt{Bool}}$$

It maps related inputs into related outputs.

Warm up exercise #2

Polymorphic equality

$$(\texttt{eq}, \texttt{eq}) \in \mathcal{R}_{\mathcal{T} \to \mathcal{T} \to \texttt{Bool}}$$

It maps related inputs into related outputs.

$$(\texttt{eq a}, \texttt{eq b}) \in \mathcal{R}_{\mathcal{T} \to \texttt{Bool}} \text{ for } (\texttt{a}, \texttt{b}) \in \mathcal{T}$$

Warm up exercise #2

Polymorphic equality

$$(\mathtt{eq}, \mathtt{eq}) \in \mathcal{R}_{\mathcal{T} \to \mathcal{T} \to \mathtt{Bool}}$$

It maps related inputs into related outputs.

$(\mathtt{eq\ \ a}, \mathtt{eq\ \ b}) \in \mathcal{R}_{\mathcal{T} \to \mathtt{Bool}}$ for $(\mathtt{a}, \mathtt{b}) \in \mathcal{T}$

$(\mathtt{eq\ \ a}, \mathtt{eq\ \ b}) \in \mathcal{R}_{\mathcal{T} \to \mathtt{Bool}}$ for any $\mathtt{a}$ and $\mathtt{b}$

Warm up exercise #2

Polymorphic equality

$(\texttt{eq a}, \texttt{eq b}) \in \mathcal{R}_{\mathcal{T} \to \texttt{Bool}}$ for any $\texttt{a}$ and $\texttt{b}$

Warm up exercise #2

Polymorphic equality

$(\texttt{eq a}, \texttt{eq b}) \in \mathcal{R}_{\mathcal{T} \to \texttt{Bool}}$ for any `a` and `b`

Function type again…

Warm up exercise #2

Polymorphic equality

$(\texttt{eq a}, \texttt{eq b}) \in \mathcal{R}_{\mathcal{T} \to \texttt{Bool}}$ for any $\texttt{a}$ and $\texttt{b}$

Function type again…

$(\texttt{eq a c}, \texttt{eq b d}) \in \mathcal{R}_{\texttt{Bool}}$ for $(\texttt{c}, \texttt{d}) \in \mathcal{T}$

Warm up exercise #2

Polymorphic equality

$(\texttt{eq a}, \texttt{eq b}) \in \mathcal{R}_{\mathcal{T} \to \texttt{Bool}}$ for any $\texttt{a}$ and $\texttt{b}$

Function type again…

$(\texttt{eq a c}, \texttt{eq b d}) \in \mathcal{R}_{\texttt{Bool}}$ for $(\texttt{c}, \texttt{d}) \in \mathcal{T}$

$(\texttt{eq a c}, \texttt{eq b d}) \in \mathcal{R}_{\texttt{Bool}}$ for any $\texttt{c}$ and $\texttt{d}$

Warm up exercise #2

Polymorphic equality

$(\texttt{eq a}, \texttt{eq b}) \in \mathcal{R}_{\mathcal{T} \to \texttt{Bool}}$ for any $\texttt{a}$ and $\texttt{b}$

Function type again…

$(\texttt{eq a c}, \texttt{eq b d}) \in \mathcal{R}_{\texttt{Bool}}$ for $(\texttt{c}, \texttt{d}) \in \mathcal{T}$

$(\texttt{eq a c}, \texttt{eq b d}) \in \mathcal{R}_{\texttt{Bool}}$ for any $\texttt{c}$ and $\texttt{d}$

$\texttt{eq a c = eq b d}$ for any $\texttt{a}$, $\texttt{b}$, $\texttt{c}$, and $\texttt{d}$

Not a very useful operator

Let's get back to the original example!

Back to the example

```
f :: forall A . [A] -> [A]
```

Back to the example

`f :: forall A . [A] -> [A]`

means

$$(\mathtt{f}, \mathtt{f}) \in \mathcal{R}_{\forall \mathcal{A}.[\mathcal{A}] \to [\mathcal{A}]}$$

and we can pick any relation for $\mathcal{A}$.

Back to the example

```
f :: forall A . [A] -> [A]
```

means

$$(\texttt{f}, \texttt{f}) \in \mathcal{R}_{\forall \mathcal{A}.[\mathcal{A}] \to [\mathcal{A}]}$$

and we can pick any relation for $\mathcal{A}$.

let's pick one that represents a function: g.

$$(\texttt{f}, \texttt{f}) \in \mathcal{R}_{[g] \to [g]}$$

Back to the example

$$(\mathtt{f}, \mathtt{f}) \in \mathcal{R}_{[g] \to [g]}$$

Back to the example

$$(\mathtt{f}, \mathtt{f}) \in \mathcal{R}_{[g] \to [g]}$$

$(\mathtt{f\ a}, \mathtt{f\ b}) \in \mathcal{R}_{[g]}$ for $(\mathtt{a}, \mathtt{b}) \in \mathcal{R}_{[g]}$

Back to the example

$$(\mathtt{f}, \mathtt{f}) \in \mathcal{R}_{[g] \to [g]}$$

$(\mathtt{f\ a}, \mathtt{f\ b}) \in \mathcal{R}_{[g]}$ for $(\mathtt{a}, \mathtt{b}) \in \mathcal{R}_{[g]}$

$\mathcal{R}_{[g]}$ relates `a` with `map g a`

Back to the example

$$(\mathtt{f}, \mathtt{f}) \in \mathcal{R}_{[g] \to [g]}$$

$$(\mathtt{f\ a}, \mathtt{f\ b}) \in \mathcal{R}_{[g]} \text{ for } (\mathtt{a}, \mathtt{b}) \in \mathcal{R}_{[g]}$$

$\mathcal{R}_{[g]}$ relates `a` with `map g a`

$$(\mathtt{f\ a}, \mathtt{f\ (map\ g\ a)}) \in \mathcal{R}_{[g]}$$

Back to the example

$$(\mathtt{f}, \mathtt{f}) \in \mathcal{R}_{[g] \to [g]}$$

$(\mathtt{f\ a}, \mathtt{f\ b}) \in \mathcal{R}_{[g]}$ for $(\mathtt{a}, \mathtt{b}) \in \mathcal{R}_{[g]}$

$\mathcal{R}_{[g]}$ relates `a` with `map g a`

$$(\mathtt{f\ a}, \mathtt{f\ (map\ g\ a)}) \in \mathcal{R}_{[g]}$$

$\mathcal{R}_{[g]}$ in action again…

Back to the example

$$(\mathtt{f},\mathtt{f}) \in \mathcal{R}_{[g]\to[g]}$$

$(\mathtt{f\ a},\mathtt{f\ b}) \in \mathcal{R}_{[g]}$ for $(\mathtt{a},\mathtt{b}) \in \mathcal{R}_{[g]}$

$\mathcal{R}_{[g]}$ relates `a` with `map g a`

$$(\mathtt{f\ a},\mathtt{f\ (map\ g\ a)}) \in \mathcal{R}_{[g]}$$

$\mathcal{R}_{[g]}$ in action again…

`map g (f a) = f (map g a)`

Back to the example

```
f :: forall A . [A] -> [A]
```

yields

```
f (map g a) = map g (f a)
```

or

```
f . (map g) = (map g) . f
```

Our first free theorem!

Wait, free?!

We worked pretty hard…

Free theorems

```
f :: forall A . [A] -> [A]
```

```
f . (map g) = (map g) . f
```

Free theorems

```
f :: forall A . [A] -> [A]

f . (map g) = (map g) . f

tail . (map g) = (map g) . tail
```

Free theorems

```
f :: forall A . [A] -> [A]

f . (map g) = (map g) . f

tail . (map g) = (map g) . tail

reverse . (map g) = (map g) . reverse
```

Free theorems

```
f :: forall A . [A] -> [A]

f . (map g) = (map g) . f

tail . (map g) = (map g) . tail

reverse . (map g) = (map g) . reverse

(take 5) . (map g) = (map g) . (take 5)
```

Free theorems

```
f :: forall A . [A] -> [A]

   f . (map g) = (map g) . f
tail . (map g) = (map g) . tail
reverse . (map g) = (map g) . reverse
(take 5) . (map g) = (map g) . (take 5)
(drop 3) . (map g) = (map g) . (drop 3)
```

...

More free theorems

```
(++) :: forall A . [A] -> [A] -> [A]
map f (xs ++ ys) = (map f xs) ++ (map f ys)
```

More free theorems

```
filter :: forall A.(A -> Bool) -> [A] -> [A]
(filter p).(map f) = (map f).(filter p.f)
```

Also works for `dropWhile`, `takeWhile` etc.

More free theorems

```
sort :: forall A.(A->A->Bool) -> [A] -> [A]
(map f).(sort (<)) = (sort (<)).(map f)
```

When `f` preserves `(<)`

Also works for `nub` etc.

Why bother?

Is it useful at all?

Where to go now?

Philip Wadler, (1989) Theorems for free!

John C. Reynolds, (1983) Types, abstraction and parametric polymorphism

Derek Dreyer, Parametricity and Relational Reasoning (video)

Richard Bird, Pearls of Functional Algorithm Design

Thanks!

To Małgorzata Nowak (@malgonowak), for making the slides purrific!

QUESTIONS??