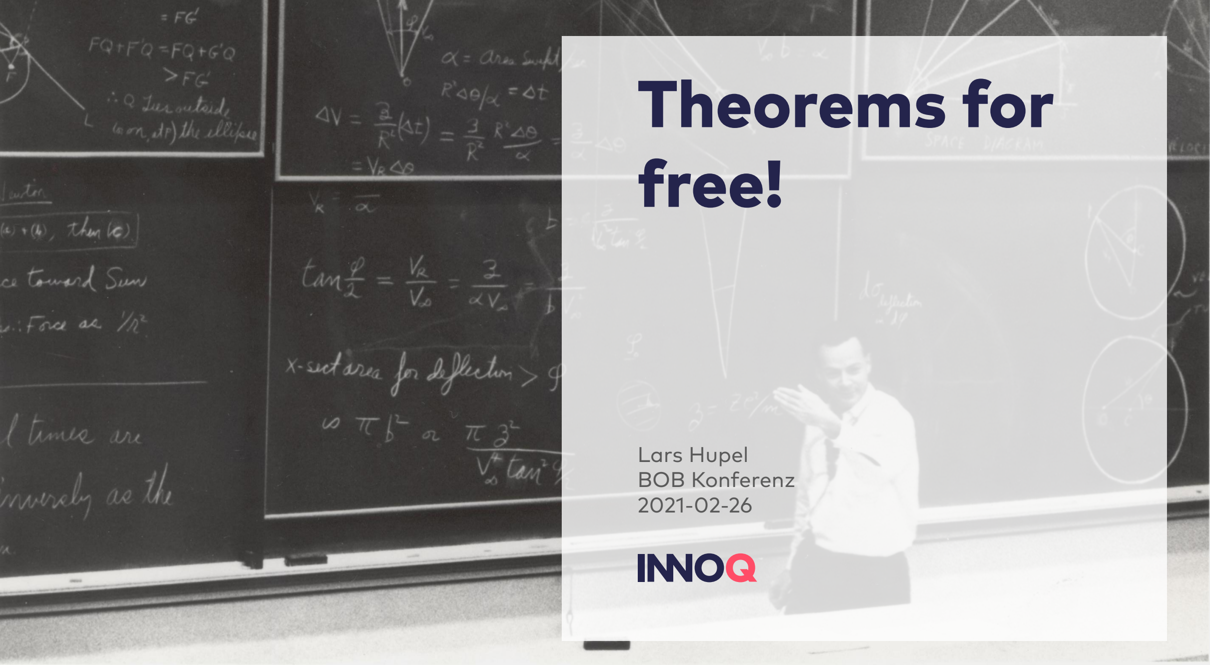


Theorems for free!

Lars Hupel
BOB Konferenz
2021-02-26

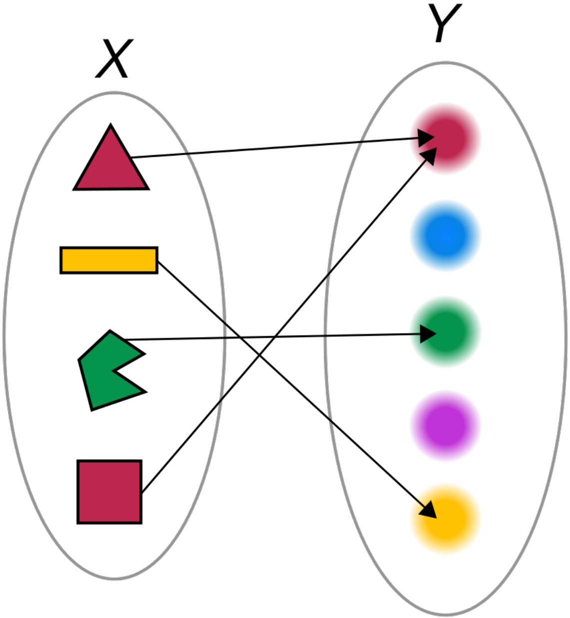
INNOQ


Let’s talk about sets

In set theory, everything[1] is a set.

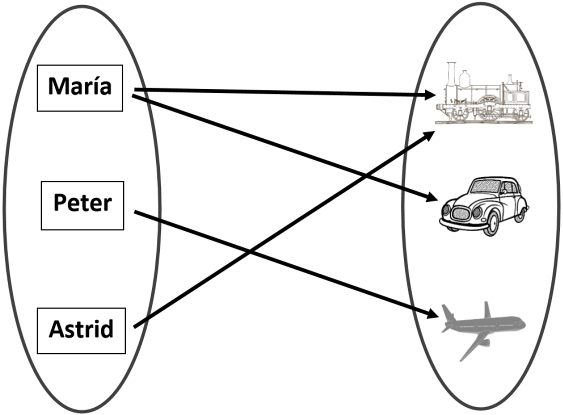
For example: $\mathbb{N} = \{0, 1, 2, \ldots\}$

[1] almost


X
Y

Functions on sets

$$f = \{({\color{red}\blacktriangle}, {\color{red}\bullet}), ({\color{yellow}\blacksquare}, {\color{yellow}\bullet}), \ldots\}$$


María
Peter
Astrid

Relations on sets

$R = \{(\text{María},$  $), (\text{María},$  $), \ldots\}$

Abstraction

Algorithms 101

Most algorithms are described in pseudocode.

procedure BellmanKalaba(G, u, l, p)
    for all $v \in V(G)$ **do**
        $l(v) \leftarrow \infty$
    end for
    $l(u) \leftarrow 0$
    $\triangleright$ and so on ...
end procedure

Why pseudocode?

Pseudocode is nice because it abstracts away implementation details.

Why pseudocode?

Pseudocode is nice because it abstracts away implementation details.

`HashSet? List? Array?`

Why pseudocode?

Pseudocode is nice because it abstracts away implementation details.

~~`HashSet`? `List`? `Array`?~~ Sets!

Getting real

At some point, we need to implement algorithms.

How can we justify replacing abstract sets with concrete lists?

Abstraction function

$$
\begin{aligned}
\alpha &:: \mathtt{List}\ a \to \mathtt{Set}\ a \\
\alpha([]) &= \varnothing \\
\alpha(x : xs) &= \{x\} \cup \alpha(xs)
\end{aligned}
$$

Abstraction function

$$\alpha :: \texttt{List}\ a \to \texttt{Set}\ a$$
$$\alpha([]) = \varnothing$$
$$\alpha(x : xs) = \{x\} \cup \alpha(xs)$$

Abstraction function

$$\alpha :: \texttt{List}\ a \to \texttt{Set}\ a$$

$$\alpha([]) = \varnothing$$

$$\alpha(x : xs) = \{x\} \cup \alpha(xs)$$

Example

$$\alpha([1, 2]) = \{1, 2\}$$

$$\alpha([2, 2, 1]) = \{1, 2\}$$

Remove the minimum

$$\begin{tikzcd} S \arrow[r, "f"] & f(S) \\ xs \arrow[u, dashed, "\alpha"] \arrow[r, "g"] & g(xs) \arrow[u, dashed, "\alpha"'] \end{tikzcd}$$

Remove the minimum

$$\begin{tikzcd} \{1,2,3\} \arrow[r, "f"] & \{2,3\} \\ {[3,2,1]} \arrow[u, dashed, "\alpha"] \arrow[r, "g"] & {[3,2]} \arrow[u, dashed, "\alpha"'] \end{tikzcd}$$

$$f(S) = S \setminus \min(S)$$
$$g(xs) = ?$$

Remove the minimum

$$\begin{tikzcd}
\{1,2,3\} \arrow[r, "f"] & \{2,3\} \\
{[3,2,1,1]} \arrow[u, dashed, "\alpha"] \arrow[r, "g"] & {[3,2]} \arrow[u, dashed, "\alpha"']
\end{tikzcd}$$

$$f(S) = S \setminus \min(S)$$

$$g(xs) = $$

Remove the minimum

$$
\begin{tikzcd}
\varnothing \arrow[r, "f"] & \varnothing \\
{[\,]} \arrow[u, dashed, "\alpha"] \arrow[r, "g"] & {[\,]} \arrow[u, dashed, "\alpha"']
\end{tikzcd}
$$

$$f(S) = S \setminus \min(S)$$
$$g(xs) = ?$$

Correspondence

$g :: \texttt{List}\ a \to \texttt{List}\ b$ is a valid implementation of $f :: \texttt{Set}\ a \to \texttt{Set}\ b$
if and only if:

Correspondence

$g :: \texttt{List}\ a \to \texttt{List}\ b$ is a valid implementation of $f :: \texttt{Set}\ a \to \texttt{Set}\ b$
if and only if:

- for every list *xs*,

Correspondence

g :: `List` $a \to$ `List` b is a valid implementation of f :: `Set` $a \to$ `Set` b if and only if:

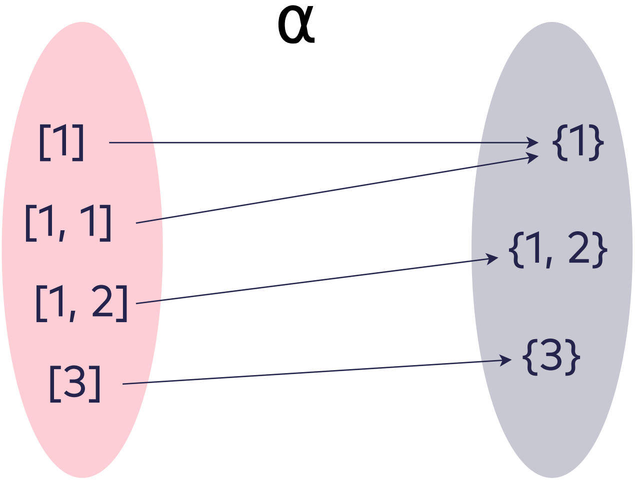
- for every list xs,
- it holds that: $\alpha(g(xs)) = f(\alpha(xs))$.

Correspondence

$g :: \texttt{List}\ a \to \texttt{List}\ b$ is a valid implementation of $f :: \texttt{Set}\ a \to \texttt{Set}\ b$
if and only if:

- for every list xs,
- it holds that: $\alpha(g(xs)) = f(\alpha(xs))$.

We say that g refines f.



α
[1]
[1, 1]
[1, 2]
[3]
{1}
{1, 2}
{3}

g refines f because related inputs are mapped to related outputs.

$$\forall x, y.\ (x, y) \in \alpha \implies (g\,x, f\,y) \in \alpha$$

If α is injective, we can use this to automatically define g.

Challenges

There's no free lunch.

- How to define "Pick $x \in S$"?
- What if α is not injective?
- What if α is partial?

Use cases

- Program refinement
- Abstract interpretation
- Parametricity

Use cases

- Program refinement
- Abstract interpretation
- Parametricity

Parametricity

Haskell folklore says:

The more type variables, the merrier!

```haskell
data Lens s a = Lens
  { getter :: s -> a
  , setter :: a -> s -> s }
```

```haskell
type Lens s t a b =
  Functor f =>
    (a -> f b) ->
    s -> f t
```

Types are sets

$$
\begin{aligned}
[\![\texttt{Bool}]\!] &= \{\texttt{True}, \texttt{False}\} \\
[\![\texttt{Integer}]\!] &= \{\ldots, -2, -1, 0, 1, 2, \ldots\} \\
[\![(a, b)]\!] &= [\![a]\!] \times [\![b]\!] \\
[\![a \to b]\!] &= \text{the set of all functions from } [\![a]\!] \text{ to } [\![b]\!]
\end{aligned}
$$

Types are relations

We can assign every type t a relation rel_t.

Types are relations

We can assign every type t a relation rel_t.

This relation will relate values of $[\![t]\!]$: $\mathrm{rel}_t \subseteq [\![t]\!] \times [\![t]\!]$

The parametricity theorem

If t is a closed term of type T, then $(t, t) \in \mathrm{rel}_T$.

The parametricity theorem

If t is a closed term of type T, then $(t, t) \in \mathrm{rel}_T$.

In other words: every term is related to itself

Let's say we have a function on lists.

```
frobnicate :: List a -> List a
```

Let's say we have a function on lists.

```
frobnicate :: List a -> List a
```

Parametricity states:

$$(\texttt{frobnicate}, \texttt{frobnicate}) \in$$ 

Let's say we have a function on lists.

```
frobnicate :: List a -> List a
```

Parametricity states:

$$(\texttt{frobnicate}, \texttt{frobnicate}) \in$$ 

We can prove:

$$\texttt{frobnicate}\ (\texttt{map}\ g\ \mathrm{xs}) = \texttt{map}\ g\ (\texttt{frobnicate}\ \mathrm{xs})$$

Now what?


BEFORE


AFTER

Reasoning about types

Motto: Functions with type variables ...

- don't know anything
- can't do much

In practise

The second `Functor` law is redundant.

It is sufficient to prove that `fmap id = id`.

Free Theorems!

Please enter a (polymorphic) type, e.g. "(a -> Bool) -> [a] -> [a]":

(a -> Bool) -> [a] -> [a]

Please choose a sublanguage of Haskell:

no bottoms (hence no general recursion and no selective strictness)

inequational theorems (only relevant in a language with bottoms)

hide type instantiations in the theorem presentation

The Free Theorem

```
forall t1,t2 in TYPES, R in REL(t1,t2).
 forall p :: t1 -> Bool.
  forall q :: t2 -> Bool.
   (forall (x, y) in R. p x = q y)
   ==> (forall (z, v) in lift{[]}(R). (f p z, f q v) in lift{[]}(R))
```

The Free Theorem

with all permissable relation variables reduced to functions

```
forall t1,t2 in TYPES, g :: t1 -> t2.
 forall p :: t1 -> Bool.
  forall q :: t2 -> Bool.
   (forall x :: t1. p x = q (g x))
   ==> (forall y :: [t1]. map g (f p y) = f q (map g y))
```

Another free theorem

A function with type `(a -> b) -> [a] -> [b]` is either

1. `map`, or
2. `map` with rearrangements

Restrictions

$\perp$ destroys everything[2]

[2] not everything

Extensions

We have ignored classes (so far) because they complicate things.

Extensions

We have ignored classes (so far) because they complicate things.

Classes can be modelled as dictionaries with (potentially) rank-2 types

Q & A

INNOQ
www.innoq.com

Lars Hupel

lars.hupel@innoq.com

@larsr_h

LARS HUPEL

Senior Consultant
innoQ Deutschland GmbH

Lars is known as one of the founders of the Typelevel initiative which is dedicated to providing principled, type-driven Scala libraries in a friendly, welcoming environment. A frequent conference speaker, they are active in the open source community, particularly in Scala.

Credits

- John C. Reynolds: https://commons.wikimedia.org/w/index.php?title=File:Reynolds_John_small.jpg&oldid=452226049, Andrej Bauer, CC-BY-SA 2.5
- Philip Wadler: https://commons.wikimedia.org/w/index.php?title=File:Wadler2.JPG&oldid=262214892, Clq, CC-BY 3.0
- Function: https://commons.wikimedia.org/w/index.php?title=File:Function_color_example_3.svg&oldid=321533277, Wvbailey, CC-BY-SA 3.0
- Relation: https://commons.wikimedia.org/w/index.php?title=File:Representative_example_of_a_mathematical_correspondence.png&oldid=505302140, Rafael Cabanillas Murillo, CC-BY-SA 4.0
- Free Theorems: https://free-theorems.nomeata.de/, Joachim Breitner et al.
- Feynman with blackboard: https://commons.wikimedia.org/wiki/File:HD.3A.053_(10481714045).jpg
- Pseudocode: http://tug.ctan.org/macros/latex/contrib/algorithmicx/algorithmicx.pdf