Logarithm and Program Testing

Favonia, 2022.03.25 j.w.w. Zhuyang Wang

Polymorphism

Parametric polymorphism (*e.g.*, ML, Haskell)

Polymorphism

Parametric polymorphism (*e.g.*, ML, Haskell)

```
map : (a -> b) -> list(a) -> list(b)
```

Polymorphism

Parametric polymorphism (*e.g.*, ML, Haskell)

```
map : (a -> b) -> list(a) -> list(b)
```

```
map (*2) [1,2,3] = [2,4,6]
```

Polymorphism

Same program for different kinds of data

```
map : (a -> b) -> list(a) -> list(b)
```

```
map (not) [true,false] = [false,true]
```

Polymorphism

Same program for different kinds of data

```
map : (a -> b) -> list(a) -> list(b)
```

Question: how can we test this function?

```
proj : (a * a) -> a
```

```
map : (a -> b) -> list(a) -> list(b)
```

Question: how can we test this function?

```
proj : (a * a) -> a
```

There are only two possibilities!

Question: how can we test this function?

`proj : (a * a) -> a`

There are only two possibilities!

$$\mathtt{proj}_1\ (x,y) = x$$

$$\mathtt{proj}_2\ (x,y) = y$$

proj : (α * α) -> α

There are only two possibilities!

proj$_1$ (x,y) = x

proj$_2$ (x,y) = y

proj (true,false) = ?

```
proj : (a * a * a) -> a
```

There are only three possibilities!

$\texttt{proj}_1\ \texttt{(x,y) = x}$

$\texttt{proj}_2\ \texttt{(x,y) = y}$

`proj (true,false) = ?`

```
proj : (a * a * a) -> a
```

There are only three possibilities!

$$\mathtt{proj}_1\ \mathtt{(x,y,z)\ =\ x}$$

$$\mathtt{proj}_2\ \mathtt{(x,y,z)\ =\ y}$$

$$\mathtt{proj}_3\ \mathtt{(x,y,z)\ =\ z}$$

```
proj : (a * a * a) -> a
```

There are only three possibilities!

$$\mathtt{proj}_1\ \mathtt{(x,y,z)} = \mathtt{x}$$

$$\mathtt{proj}_2\ \mathtt{(x,y,z)} = \mathtt{y}$$

$$\mathtt{proj}_3\ \mathtt{(x,y,z)} = \mathtt{z}$$

```
proj (0,1,2) = ?
```

```
proj : (a * a * ... * a) -> a
```

There are exactly n possibilities!

```
proj : (a * a * ... * a) -> a
```

There are exactly n possibilities!

```
proj (0,1,2,...,n-1) = ?
```

```
proj : (a * a * ... * a) -> a
```

There are exactly n possibilities!

```
proj (0,1,2,...,n-1) = ?
```

Any type with n distinct values works

```
proj : (a * a * ... * a) -> a
```

There are exactly n possibilities!

$$\log_a(a * a * \ldots * a) = \log_a(a^n) = n$$

```
proj : (a * a * ... * a) -> a
```

There are exactly n possibilities!

$$\log_a(a \ast a \ast \ldots \ast a) = \log_a(a^n) = n$$

counting elements of type a in the input

```
f : α(a) -> H(a)
```

```
f : α(a) -> H(a)
```

one of the good types for testing*

$$\log_a \alpha(a)$$

* see our paper for caveats

```
f : α(a) -> H(a)
```

one of the good types for testing*

$$\log_a \alpha(a)$$

testing is still complete**

* see our paper for caveats

** the empty type might need to be separately tested

2010 *“Testing Polymorphic Properties”*

by Bernardy, Jansson, and Claessen

`f : (F(a) -> a) * (G(a) -> K)) -> H(a)`

(other cases manually massaged into this form)

2010 *“Testing Polymorphic Properties”*
by Bernardy, Jansson, and Claessen

`f : (F(a) -> a) * (G(a) -> K)) -> H(a)`

(other cases manually massaged into this form)

2017 **Liyao Xia wrote a Haskell library roughly based on the above paper**

(with a logarithm-like operator in its codebase)

2010 *“Testing Polymorphic Properties”*
by Bernardy, Jansson, and Claessen

`f : (F(a) -> a) * (G(a) -> K)) -> H(a)`

(other cases manually massaged into this form)

2017 **Liyao Xia wrote a Haskell library roughly based on the above paper**

(with a logarithm-like operator in its codebase)

Logarithm-like operations have been discovered repeatedly in the literature but no one connected logarithm to testing

e.g., [Abbott et al. 2003; Altenkirch et al. 2015]

```
f : (a -> a) * a -> a
```

```
f : (a -> a) * a -> a
```

Suppose the input is the pair (s, x)
The output must be s(s(…s(x)…))

`f : (a -> a) * a -> a`

Suppose the input is the pair (s, x)
The output must be s(s(...s(x)...))

`f ((+1), 0) = ?`

this reveals the number of s
in its output s(s(...s(x)...))

```
f : (a -> a) * a -> a
```

```
f : (a -> a) * a -> a
```

$$
\begin{aligned}
& \log_a(a^a * a) \\
={}& \log_a(a^a) + \log_a(a) \\
={}& a * \log_a(a) + 1 \\
={}& a * 1 + 1 \\
\simeq{}& a + 1
\end{aligned}
$$

```
f : (a -> a) * a -> a
```

$$
\begin{aligned}
& \log_a(a^a * a) \\
= \; & \log_a(a^a) + \log_a(a) \\
= \; & a * \log_a(a) + 1 \\
= \; & a * 1 + 1 \\
\simeq \; & a + 1
\end{aligned}
$$

Problem: a appears in the logarithm of $a^a * a$ indicating recursion

```
f : (a -> a) * a -> a
```

$$
\begin{aligned}
& \log_a(a^a * a) \\
= {} & \log_a(a^a) + \log_a(a) \\
= {} & a * \log_a(a) + 1 \\
= {} & a * 1 + 1 \\
\simeq {} & a + 1
\end{aligned}
$$

Problem: a appears in the logarithm of $a^a * a$ indicating recursion

Solution: recursive types! $\mu a.a+1$ is the naturals $\mathbb{N}$

```
f : (a -> a) * a -> a
```

$$
\begin{aligned}
& \log_a(a^a * a) \\
= {} & \log_a(a^a) + \log_a(a) \\
= {} & a * \log_a(a) + 1 \\
= {} & a * 1 + 1 \\
\simeq {} & a + 1
\end{aligned}
$$

Problem: a appears in the logarithm of $a^a * a$ indicating recursion

Solution: recursive types!

$\mu a.a+1$ is the naturals $\mathbb{N}$

Sufficient for `f ((+1), 0) = ?`

```
f : α(a) -> H(a)
```

one of the good types for testing*

$$\mu a.\log_a \alpha(a)$$

a sufficiently large type to describe all
possible ways to generate an a-element

* see our paper for remaining caveats

Theorem 1.0*

For any two functions f, g : $\alpha(a) \rightarrow H(a)$, if they agree on all inputs when type a is instantiated with $\mu a.\log_a \alpha(a)$ and with the empty type,** then they are exactly the same function!

* see our paper for omitted conditions

** see our paper for why the empty type is needed

Logarithm for All

$$\log_a(a) = 1$$

$$\log_a(1) = 0$$

$$\log_a(\alpha_1(a) * \alpha_2(a)) = \log_a(\alpha_1(a)) + \log_a(\alpha_2(a))$$

$$\log_a(\alpha_2(a)^{\alpha_1(a)}) = \alpha_1(a) * \log_a(\alpha_2(a))$$

Logarithm for All

$\log_a(a) = 1$

$\log_a(1) = 0$

$\log_a(\alpha_1(a) * \alpha_2(a)) = \log_a(\alpha_1(a)) + \log_a(\alpha_2(a))$

$\log_a(\alpha_2(a)^{\alpha_1(a)}) = \alpha_1(a) * \log_a(\alpha_2(a))$

$\log_a(b) = \log_a(0) = 0$

$\log_a(\alpha_1(a) + \alpha_2(a)) = \log_a(\alpha_1(a)) + \log_a(\alpha_2(a))$

$\log_a(\mu b ...) = ...$

Use over-approximation to cover more types

Logarithm for All

$$\log_a(a) = 1$$

$$\log_a(1) = 0$$

$$\log_a(\alpha_1(a) * \alpha_2(a)) = \log_a(\alpha_1(a)) + \log_a(\alpha_2(a))$$

$$\log_a(\alpha_2(a)^{\alpha_1(a)}) = \alpha_1(a) * \log_a(\alpha_2(a))$$

$$\log_a(b) = \log_a(0) = 0$$

$$\log_a(\alpha_1(a) + \alpha_2(a)) = \log_a(\alpha_1(a)) + \log_a(\alpha_2(a))$$

$\log_a(\mu b\ldots) = \ldots$

How about $\max(\log_a(\alpha_1(a)), \log_a(\alpha_2(a)))$?

Use over-approximation to cover more types

```
proj : (a * a) -> a
```

Theorem: instantiate a with $\mu a.\log_a(a * a) = \mu a.2 \simeq 2$

```
proj : (a * a) -> a
```

Theorem: instantiate a with $\mu a.\log_a(a \ast a) = \mu a.2 \simeq 2$

```
proj (true,true) = ?
proj (true,false) = ?
proj (false,false) = ?
proj (false,true) = ?
```

`proj : (a * a) -> a`

Theorem: instantiate a with $\mu a.\log_a(a * a) = \mu a.2 \simeq 2$

~~proj (true,true) = true~~
proj (true,false) = ?
~~proj (false,false) = false~~
proj (false,true) = ?

Logarithm is large enough to index all a-elements

The best tests should contain maximally distinct a-elements

```
map : (a -> b) -> list(a) -> list(b)
```

`map : (a -> b) -> list(a) -> list(b)`

one of the best types and inputs

$$a = b = \mathbb{N}$$

$$\mathtt{map}\ \mathtt{id}_{\mathbb{N}}\ [0,1,...,n-1] = ?$$

intuition: an evil programmer can only do these three things:
duplicating, omitting, or permuting elements in the output list
the above test case detects all possible deviations
map id [1,1,1,1] in comparison is much less useful

Theorem 2.0*

For any two functions f, g : $\alpha(a) \to H(a)$, if they agree on optimal inputs with distinct a-elements** when type a is instantiated with $\mu a.\log_a \alpha(a)$ and on all inputs with the empty type,*** then they are exactly the same function!

* see our paper for omitted conditions
** see our paper for the precise definition
*** see our paper for why the empty type is needed

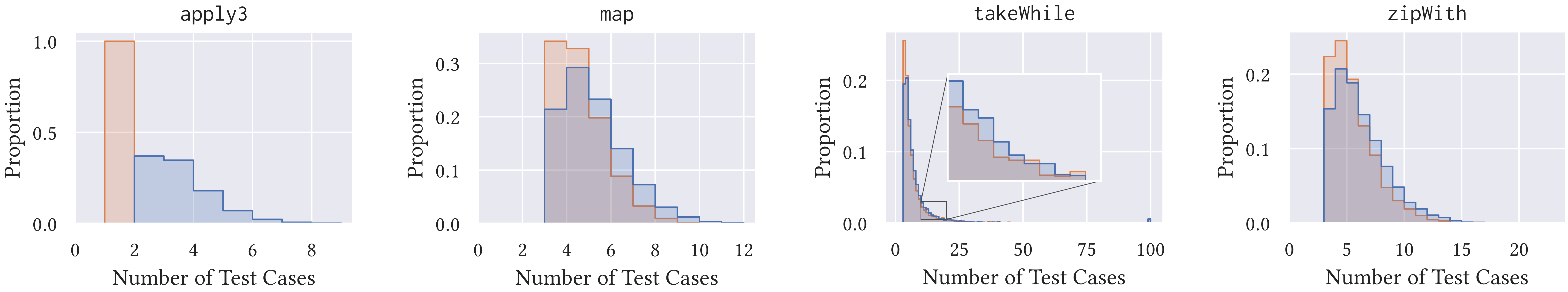
Implementation

Haskell library: `github.com/hawnzug/polycheck`

automatically specializing types and inputs

Can work with either QuickCheck or SmallCheck

Implementation



Implementation

🟧 PolyCheck with QuickCheck

🟦 Original QuickCheck

Preliminary experiments showed it requires fewer test cases to find counterexamples

Future Work

1. Incorporate information other than the typing
   e.g., `sort cmp list` expects `cmp` to form a total order

Future Work

1. Incorporate information other than the typing
   e.g., `sort cmp list` expects `cmp` to form a total order
2. Extend work to test an API, not just one function

Future Work

1. Incorporate information other than the typing

   e.g., `sort cmp list` expects `cmp` to form a total order

2. Extend work to test an API, not just one function

3. Further optimize the theorem

   e.g., for `length : list(a) -> int`
   the best choice is the unit type, not natural numbers