QuickSpec 2
(joint work with Moa)

About QuickSpec

Generates laws about functional programs by testing

- Either standalone or as a component in HipSpec

Old version had scalability problems, new design works on much harder theories!

Paper submitted to ICFP

DEMO

How it works, part 1

Enumerate terms, starting from the simplest:

- xs, ys, zs, [], xs++[], ys++[], ..., xs++(ys++zs), ..., xs++(xs++ys), ..., (xs++ys)++zs, ...

For each term, work out: *is it equal to a term we've already seen?*

- If we can *prove* it equal to a term we've already seen, using the discovered laws, discard the term
- Otherwise, if *testing* shows it's equal to a term we've already seen, we've discovered an equation!
- Otherwise, add it to the set of seen terms

t works, part 1

Knuth-Bendix completion

s, starti

QuickCheck on 1000 random test cases

- xs, ys, zs, s++[], ys++[], xs++(ys++ ..., xs++(xs++

For each te n, work out: *is qual to a term we've already seen?*

- If we can *prove* it equal t a term we've already seen, using the disco ered laws, discard the term
- Otherwise, if *testing* shows it's equal to a term we've already seen, we've discovered an equation!
- Otherwise, add it to the set of seen terms

How it works, part 1

Next term: **`xs`**

- Not equal to a seen term
- Add to seen terms!

Seen terms

`xs`

Discovered laws

How it works, part 1

Same for **`ys`**, **`zs`**, **`[]`**

Seen terms

`xs  ys  zs  []`

Discovered laws

How it works, part 1

Next term: **`[]++xs`**

Testing reveals it's equal to the already-seen term `xs`

Hooray, a law!

Seen terms

`xs` `ys` `zs` `[]`

Discovered laws

How it works, part 1

Next term: **`[]++xs`**

Testing reveals it's equal to the already-seen term `xs`

Hooray, a law!

Seen terms

```
xs   ys   zs   []
```

Discovered laws

```
[]++xs = xs
```

How it works, part 1

Next term: **`[]++ys`**

The laws imply it's equal to the already-seen term `ys`

Throw it out!

Seen terms

`xs` **`ys`** `zs` `[]`

Discovered laws

```
[]++xs = xs
```

How it works, part 1

Next term: **`xs++(ys++zs)`**

- Not equal to an already-seen term
- Add to seen terms!

Seen terms

```
xs   ys   zs   []
xs++(ys++zs)
```

Discovered laws

```
[]++xs = xs
```

How it works, part 1

Next term: **`xs++(xs++ys)`**

- Not equal to an already-seen term
- Add to seen terms!

Seen terms

```
xs  ys  zs  []
xs++(ys++zs)
xs++(xs++ys)
```

Discovered laws

```
[]++xs = xs
```

How it works, part 1

Next term: **`(xs++ys)++zs`**

Equal to `xs++(ys++zs)` by testing

Hooray, a law!

Seen terms

`xs  ys  zs  []`

`xs++(ys++zs)`

`xs++(xs++ys)`

Discovered laws

```
[]++xs = xs
```

How it works, part 1

Next term: **`(xs++ys)++zs`**

Equal to `xs++(ys++zs)` by testing

Hooray, a law!

Seen terms

```
xs  ys  zs  []
xs++(ys++zs)
xs++(xs++ys)
```

Discovered laws

```
[]++xs = xs
(xs++ys)++zs =
  xs++(ys++zs)
```

How it works, part 1

Next term: **`(xs++xs)++ys`**

Equal to `xs++(xs++ys)` by the discovered laws

Throw it out!

Seen terms

```
xs  ys  zs  []
xs++(ys++zs)
xs++(xs++ys)
```

Discovered laws

```
[]++xs = xs
(xs++ys)++zs =
  xs++(ys++zs)
```

Why is this fast?

Each term either:

- Gets thrown out without being tested
  (this is dead cheap)
- Is tested and not equal to any earlier term
  (this is dead cheap, typically < 10 tests required)
- Appears in an actual generated law
  (these terms are tested on 1000 test cases,
  but they are the very terms we want to test!)

This means we spend most of our time testing the laws we actually print out!

Schemas

We still generate a lot of similar-looking terms, like:

```
xs++(ys++zs)
ys++(xs++zs)
xs++(xs++ys)
ys++(zs++xs)
xs++(ys++xs)
zs++(xs++ys)
```

```
xs++(zs++ys)
xs++(xs++xs)
xs++(ys++ys)
ys++(xs++xs)
zs++(ys++xs)
ys++(zs++zs)
```

Surely they can't all be necessary?

Schemas

Idea 1: collect terms into *schemas*

```
?++(?++?)
```

Idea 2: discover *schema laws* and then generalise them

```
(?++?)++? = ?++(?++?)
```

A schema law holds if the *most specific instance* holds:

```
(xs++xs)++xs = xs++(xs++xs)
```

Schemas

Two-stage algorithm:

- Enumerate schemas looking for schema laws (concretely, generate the most specific instance of each schema)
- When we discover a schema law, generalise it… by generating all instances of the schemas in the law, reverting to the previous algorithm!

Gain comes because we completely avoid instantiating schemas that *don't* appear in any laws

Schemas (skip ahead a bit)

Next schema: **?++(?++?)**

Generate most specific instance: **xs++(xs++xs)**.

Not equal to an existing term.

Add to term set.

Seen terms

```
[]   xs
```

Discovered laws

```
[]++xs = xs
```

Schemas (skip ahead a bit)

Next schema: **(?++?)++?**

Generate most specific instance: **(xs++xs)++xs**.

Equal to existing term **xs++(xs++xs)**!

Seen terms

`[]  xs`

`xs++(xs++xs)`

Discovered laws

`[]++xs = xs`

Schemas

We've discovered a schema law!

(?++?)++? = ?++(?++?)

Now we generate all instances of both sides, most general instance first:

xs++(ys++zs), xs++(zs++ys), zs++(xs++ys), ..., (xs++ys)++zs, (xs++zs)++ys, (zs++xs)++ys, ...

Schemas

Terms xs++(ys++zs), zs++(xs++ys) are not equal to anything we've already seen

Seen terms

```
[]  xs
xs++(xs++xs)
xs++(ys++zs)
zs++(xs++ys) ..
```

Discovered laws

```
[]++xs = xs
```

Schemas

Next term: **`(xs++ys)++zs`**

Equal to existing term **`xs++(ys++zs)`**.

We've discovered a law!

Existing terms

```
[]  xs
xs++(xs++xs)
xs++(ys++zs)
zs++(xs++ys) ..
```

Discovered laws

```
[]++xs = xs
```

Schemas

Next term: **`(xs++ys)++zs`**

Equal to existing term **`xs++(ys++zs)`**.

We've discovered a law!

Existing terms

```
[]  xs
xs++(xs++xs)
xs++(ys++zs)
zs++(xs++ys) ..
```

Discovered laws

```
[]++xs = xs
(xs++ys)++zs =
  xs++(ys++zs)
```

Schemas

Next term: **`(zs++xs)++ys`**

Equal to existing term **`zs++(xs++ys)`** by existing law.

Throw term away!

Existing terms

```
[]  xs
xs++(xs++xs)
xs++(ys++zs)
zs++(xs++ys) ..
```

Discovered laws

```
[]++xs = xs
(xs++ys)++zs =
  xs++(ys++zs)
```

Schemas

Next schema: **?++(?++(?++?))**

Most specific instance: **xs++(xs++(xs++xs))**.

Not equal to an existing term.
Don't instantiate the schema!

Existing terms

```
[]  xs
xs++(xs++xs)
xs++(ys++zs)
zs++(xs++ys)
xs++(xs++(xs++xs))
```

Discovered laws

```
[]++xs = xs
(xs++ys)++zs =
  xs++(ys++zs)
```

Next schema:

Most specific i).

Not equal to an ex ng term.
Don't instantiate the schema!

4^4 = 256 instances
that we've avoided generating!

Existing terms

```
[]  xs
xs++(xs++xs)
xs++(ys++zs)
zs++(xs++ys)
xs++(xs++(xs++xs))
```

Discovered laws

```
[]++xs = xs
(xs++ys)++zs =
  xs++(ys++zs)
```

Schemas, a subtlety

What about laws like commutativity?

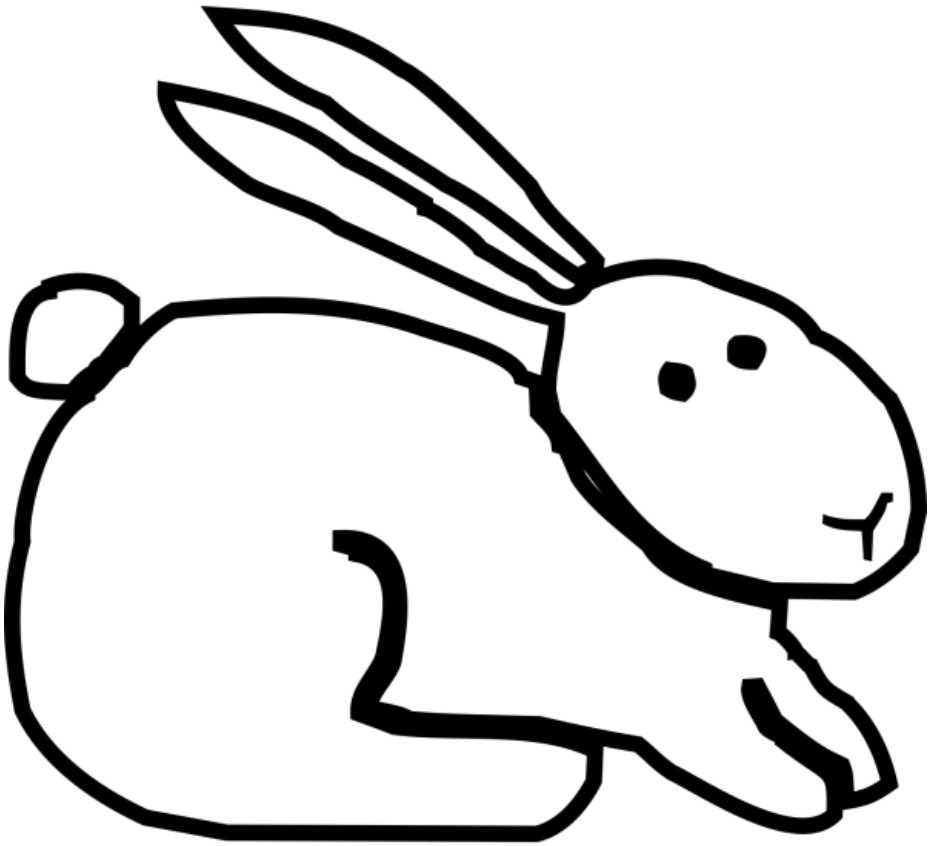
x+y = y+x

It has the same schema on both sides!

Heuristic to discover it:

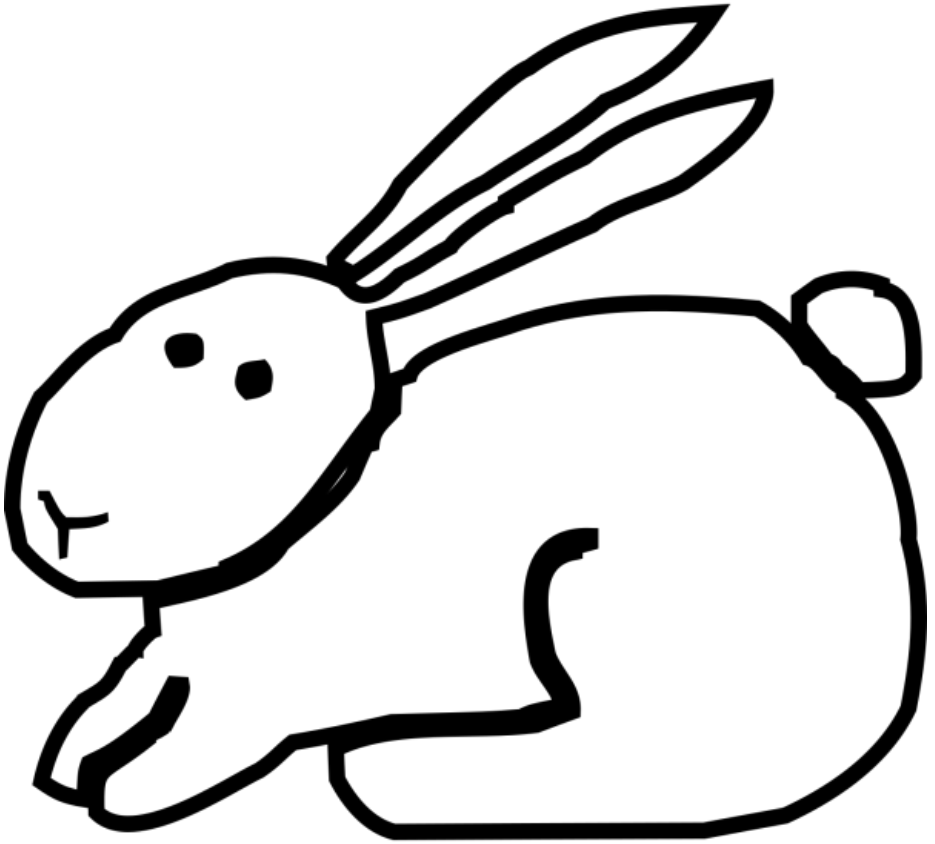
Always instantiate any schema that's below a certain (user-definable) size threshold

Functional geometry



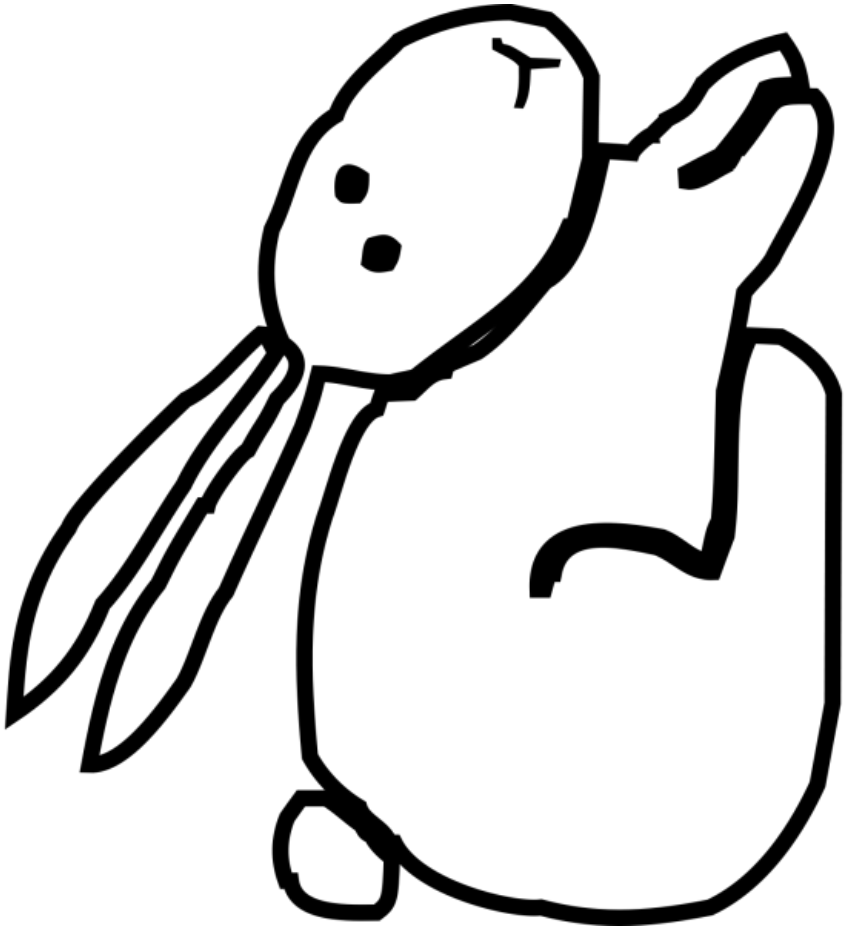
bunny

Functional geometry



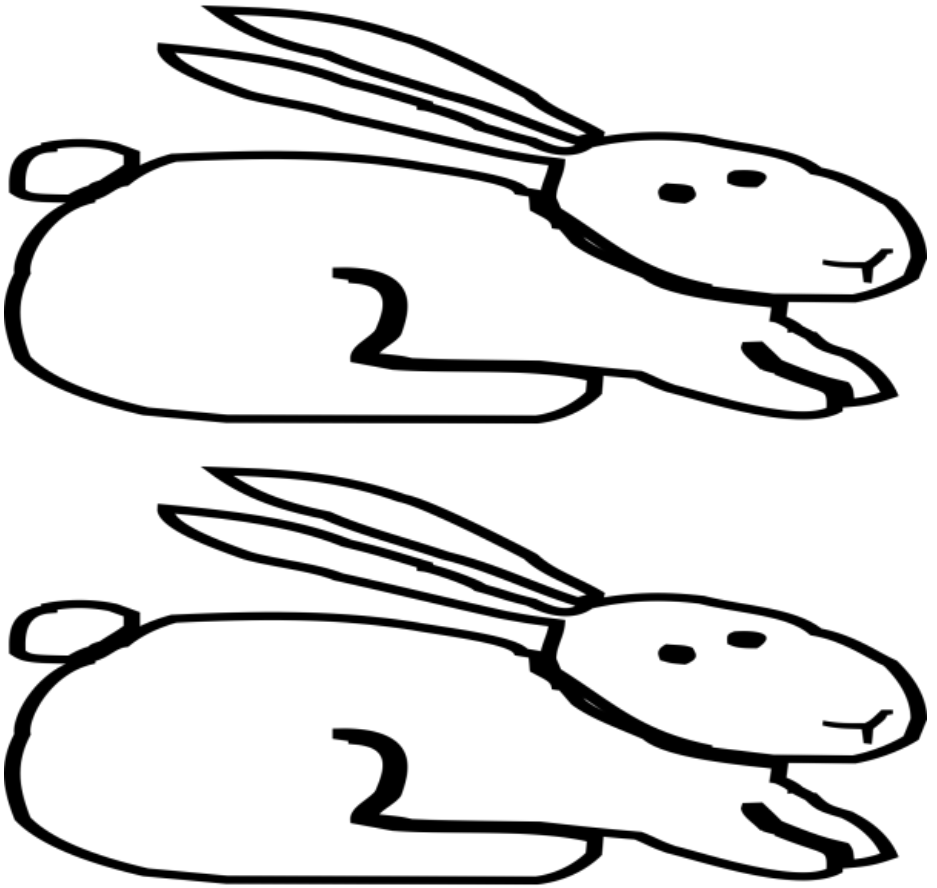
flip(bunny)

Functional geometry



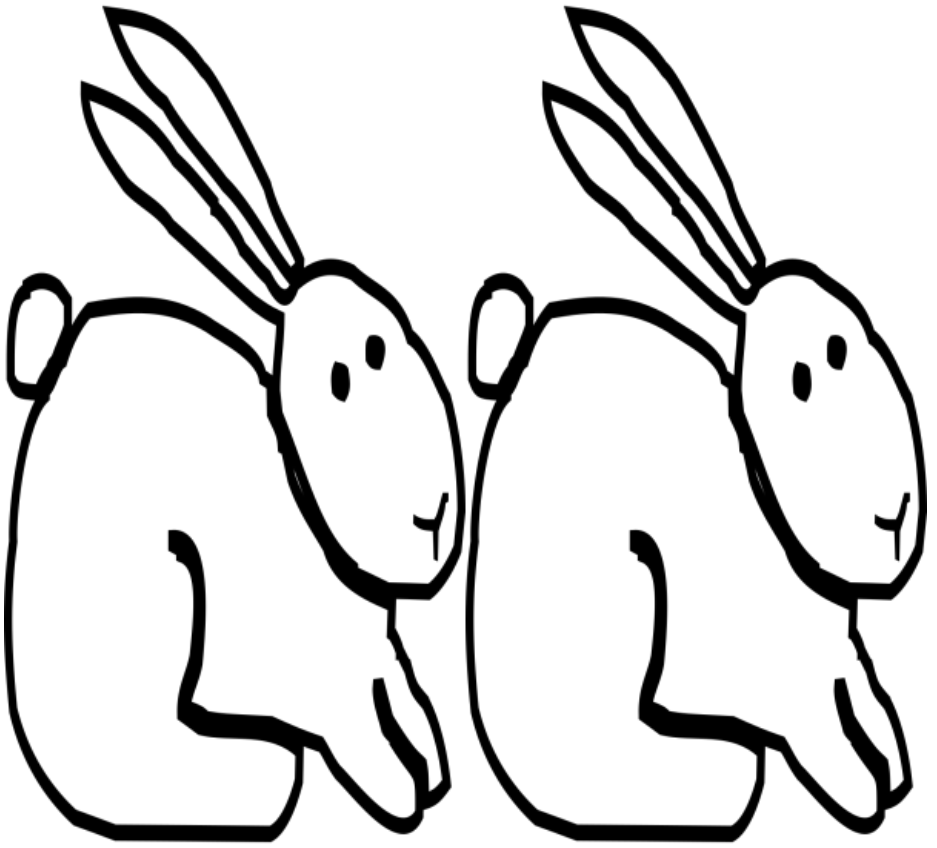
rot(bunny)

Functional geometry



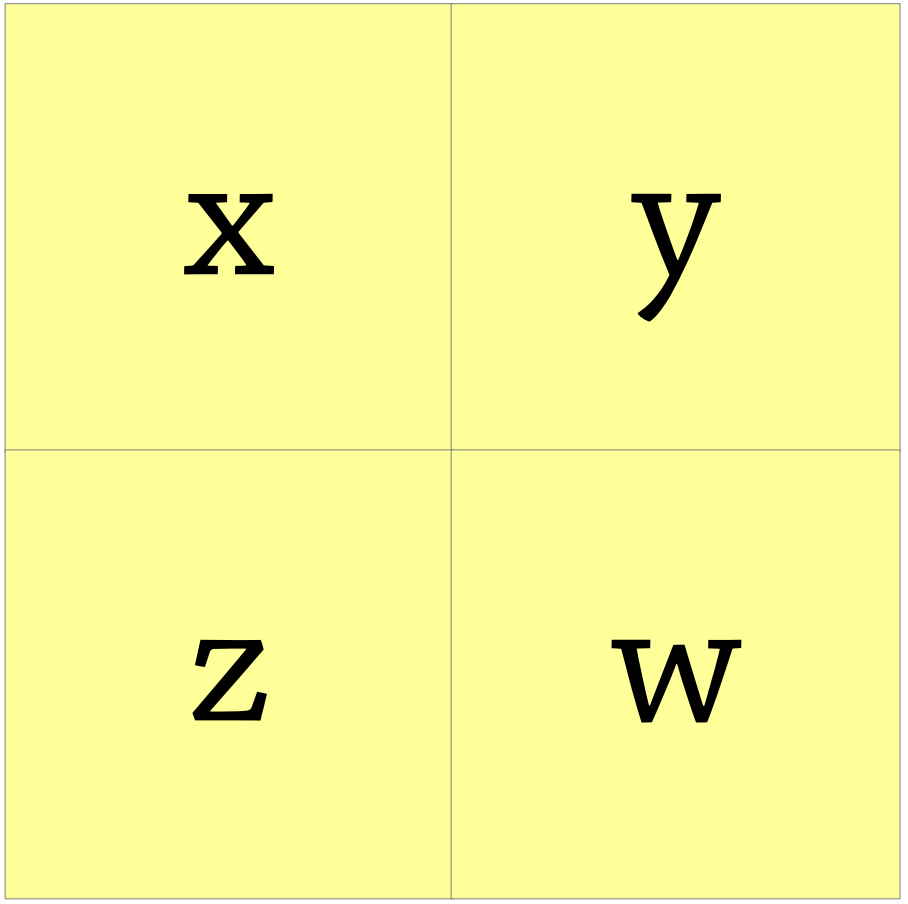
above(bunny, bunny)

Functional geometry

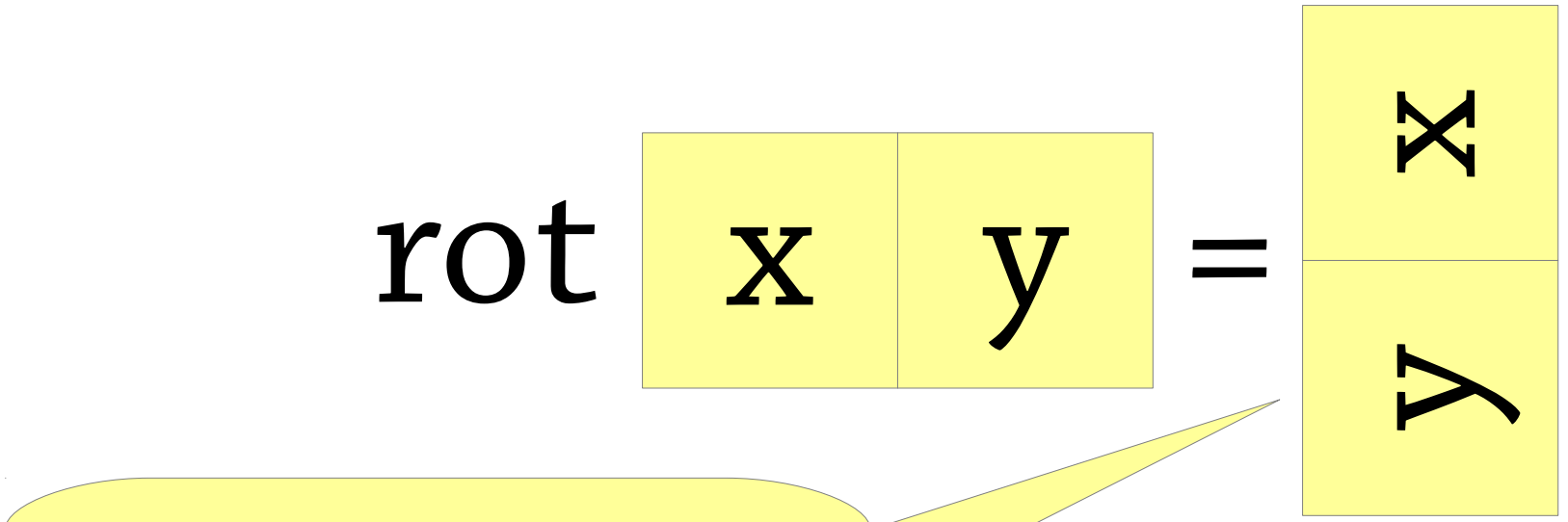


beside(bunny, bunny)

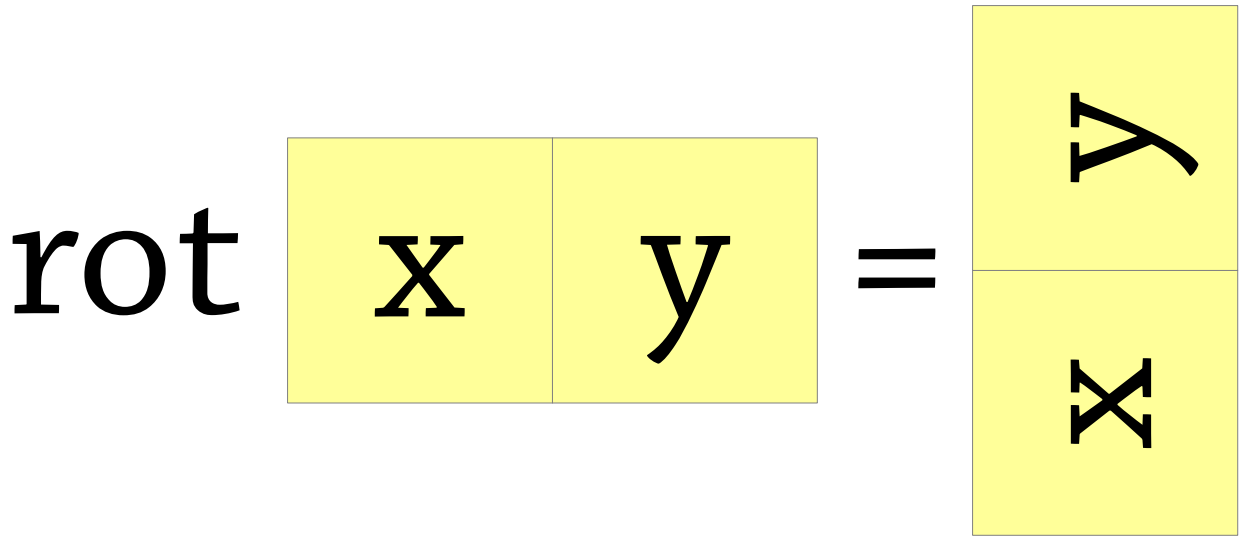
beside(above(x, z), above(y, w)) =
above(beside(x, y), beside(z, w))



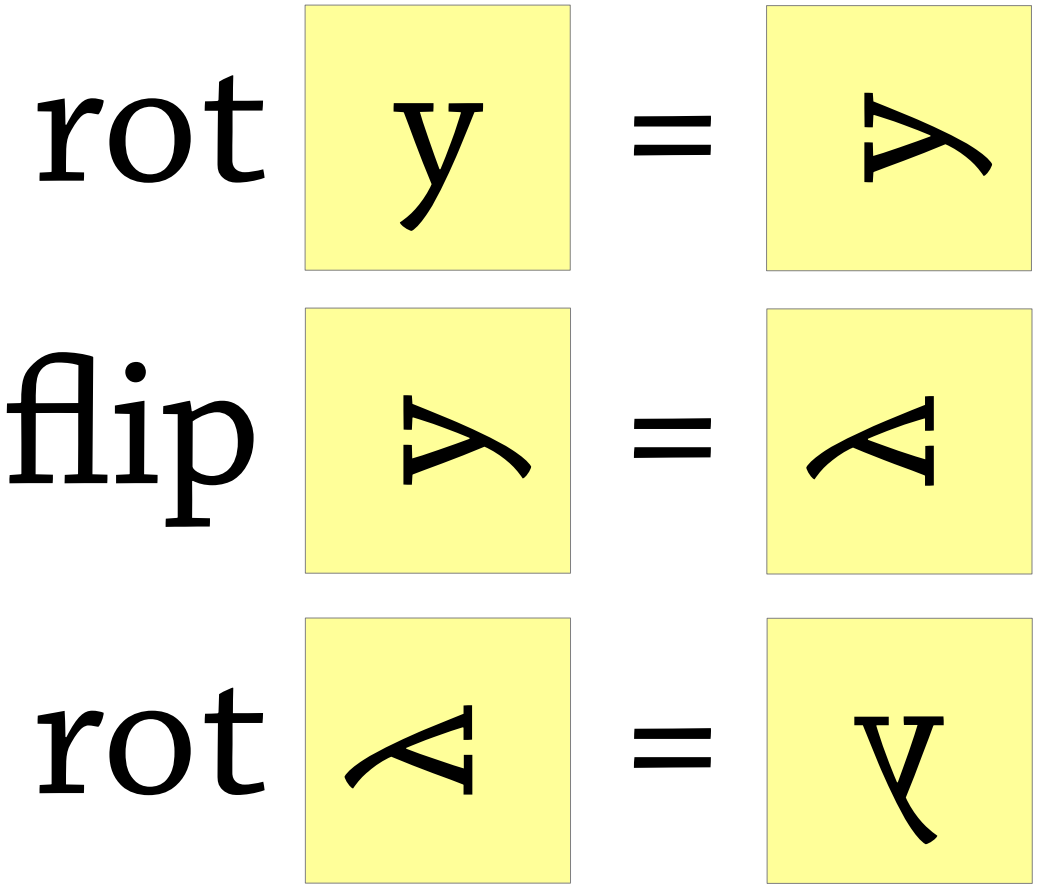
above(rot(x), rot(y)) =
rot(beside(x, y))



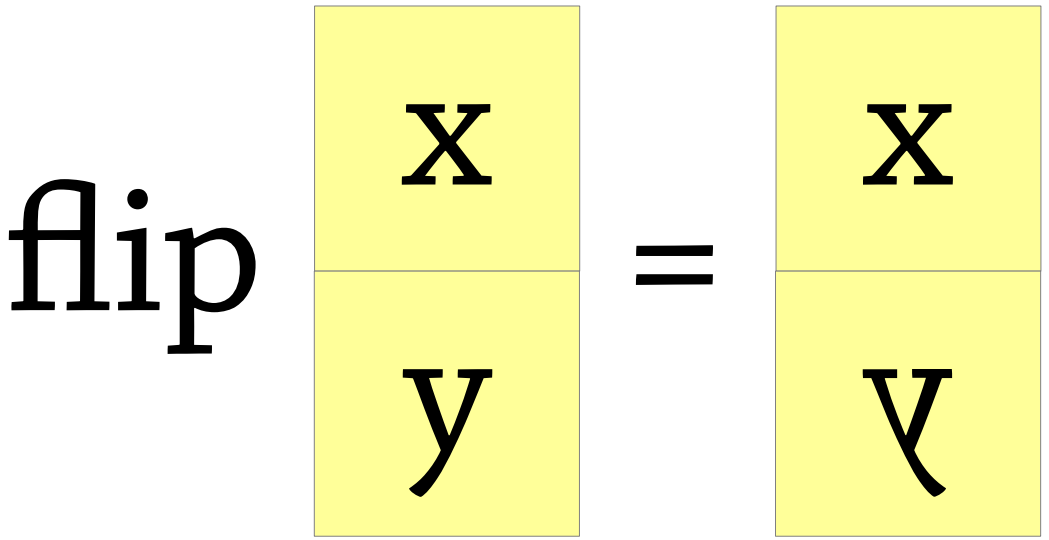
above(rot(y), rot(x)) =
rot(beside(x, y))



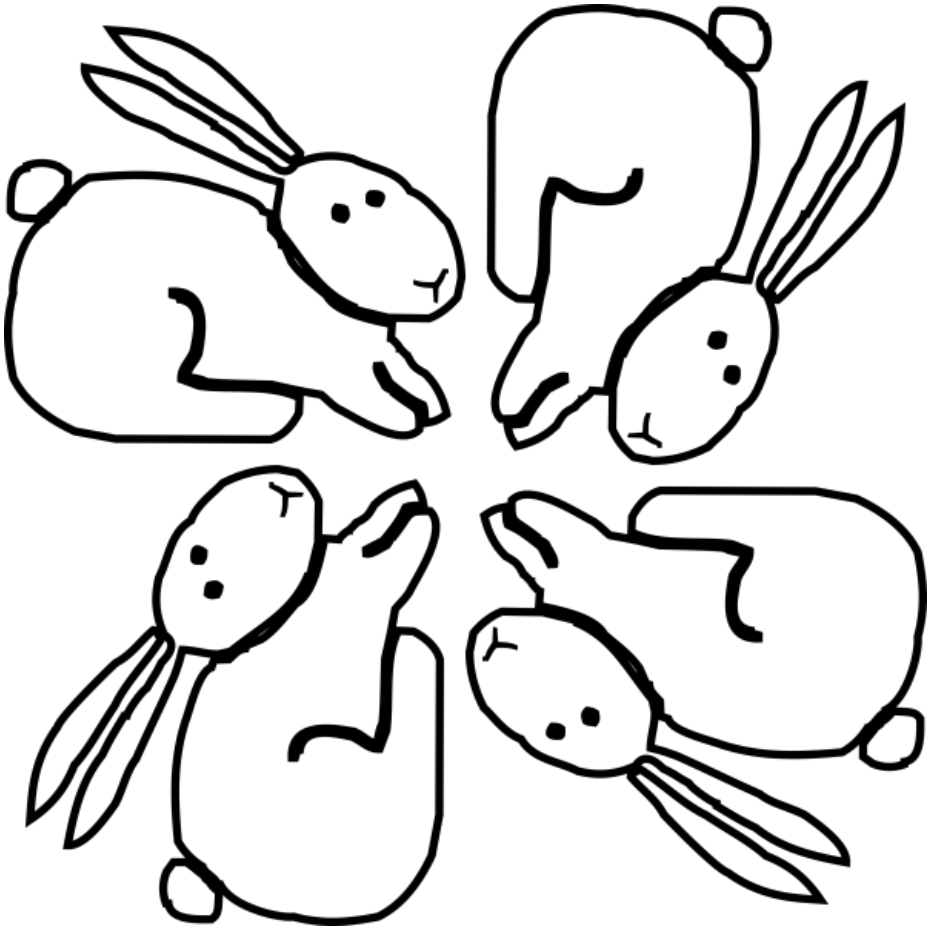
rot(flip(rot(x))) = flip(x)



above(flip(x), flip(y)) = flip(above(x, y))

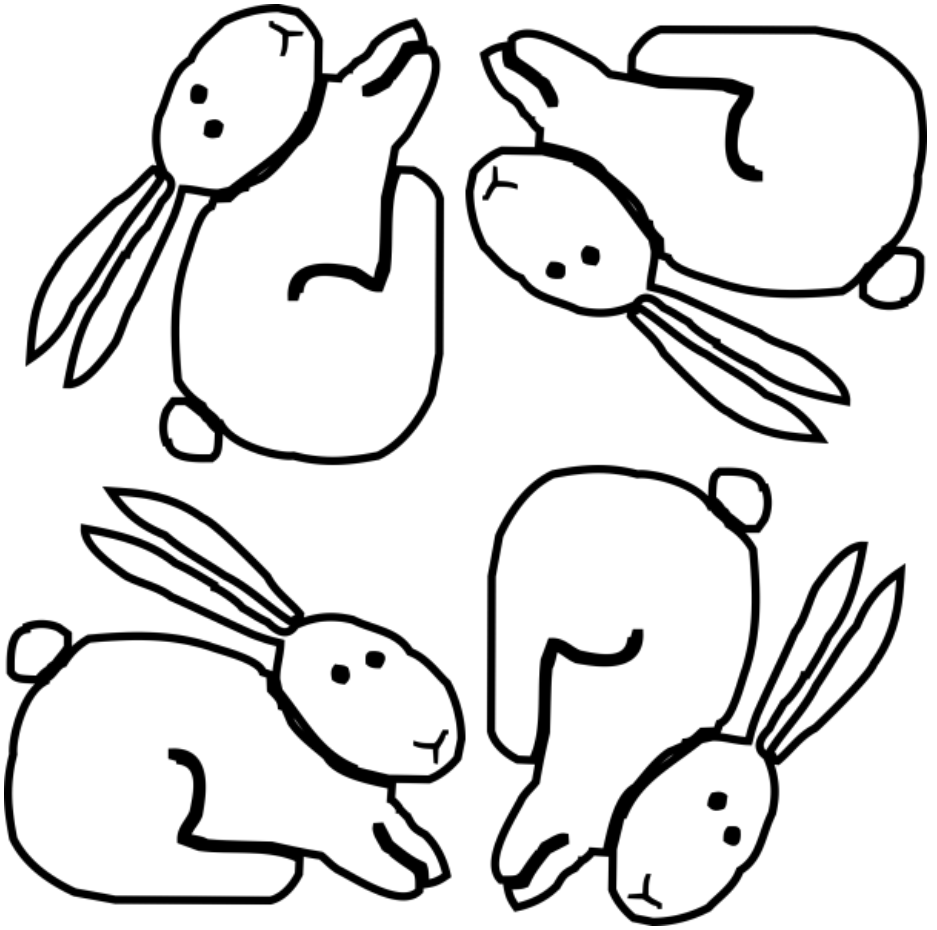


Functional geometry



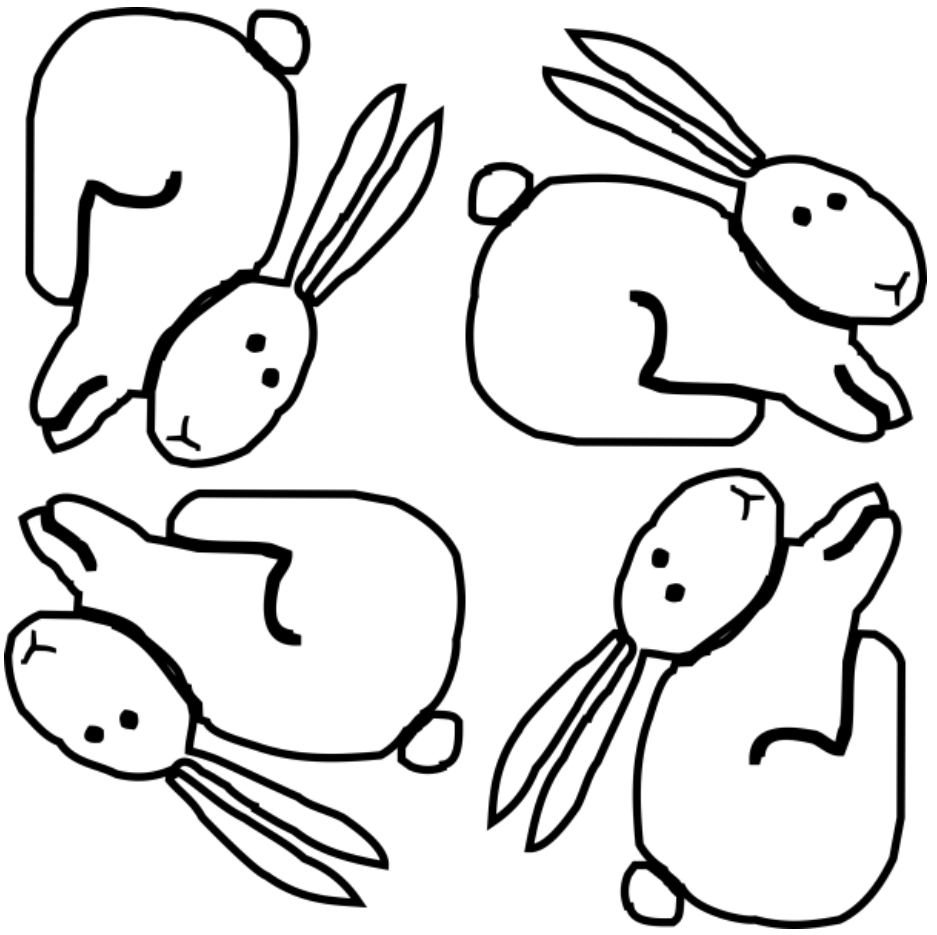
cycle(bunny)?

Functional geometry



cycle(bunny)!

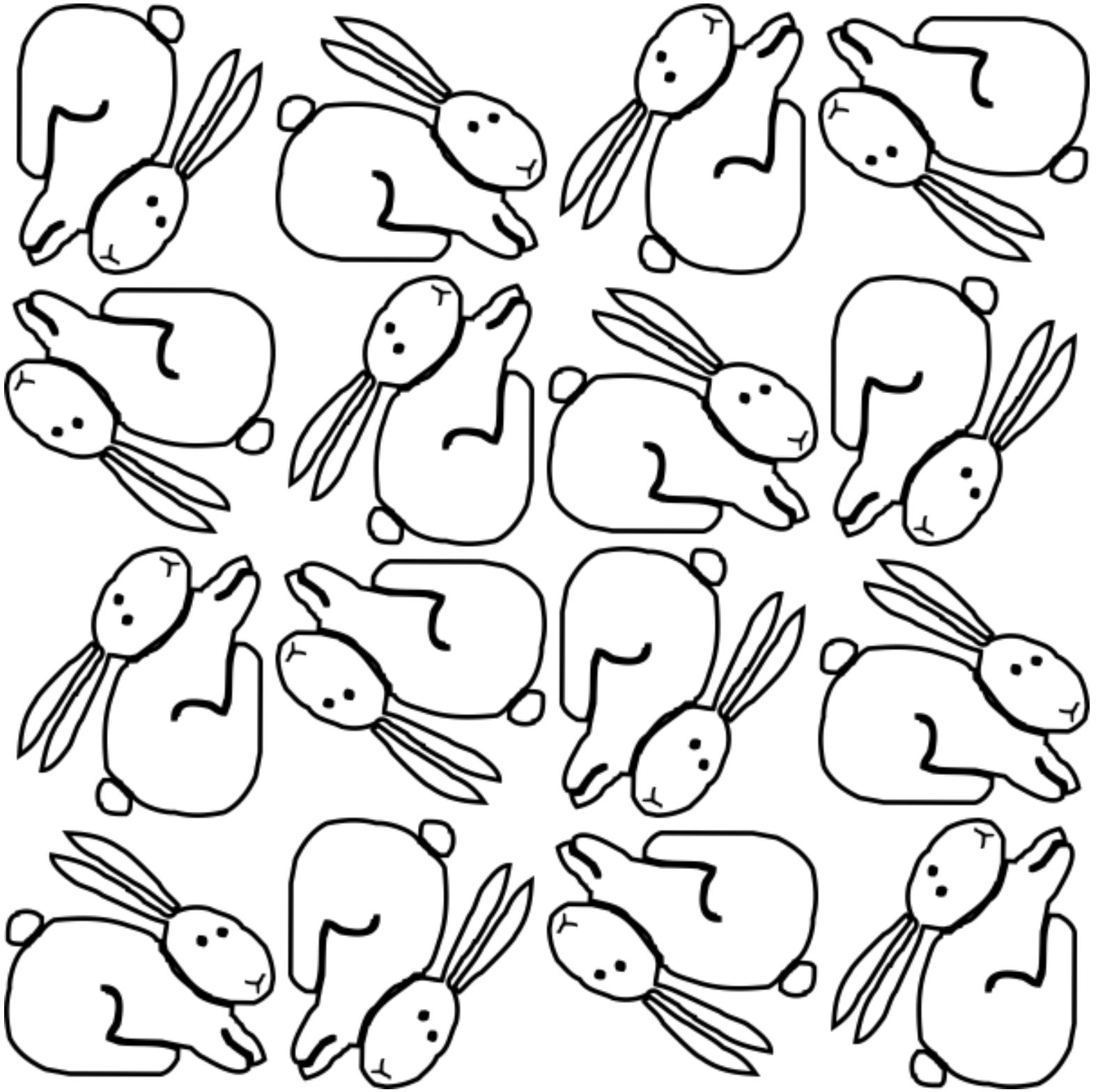
Functional geometry



rot(cycle(bunny))

cycle(cycle(bunny)) =

...



QuickSpec now

Two big scalability improvements:

- Throw away redundant terms without testing them
- Avoid generating renamings of a term

Future improvements:

- Smarter schema instantiation
  (instead of all-or-nothing like now)
- Stuff like conditional laws (easier to do now!)

Lots of fun to use!