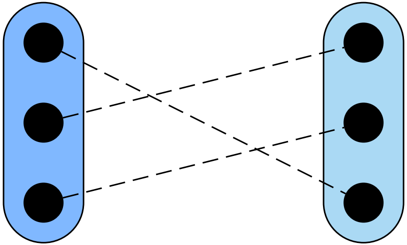
What's the Difference?

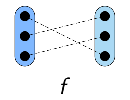
A Functional Pearl on Subtracting Bijections

Brent Yorgey, Hendrix College
Kenny Foner, University of Pennsylvania

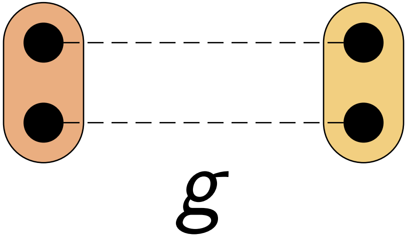
ICFP 2018 St. Louis



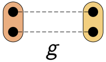
f



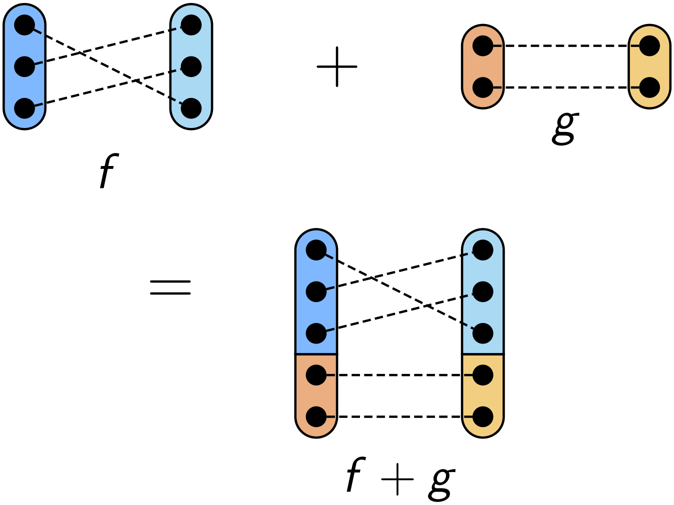
This is a bijection. It matches up the elements of these two blue sets in such a way that each element is matched with exactly one element from the other set.

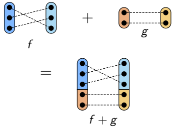


g



And here is another bijection.


f
$+$
g
$=$
$f + g$



Given these two bijections, we can add them by running them in parallel, so to speak. That is, I take the disjoint union of the dark blue and dark orange sets, and the disjoint union of the light blue and light orange sets, and I get a new bijection between these disjoint unions, which does f on one side and g on the other.

Ground rules

Ground rules

I need to stop at this point to establish some ground rules for the rest of my talk.

1. “type” = “set”

1. "type" = "set"

Rule number 1: types and sets are the same thing. I am going to use these words interchangeably. Rule number 2: everything is finite. OK? After my talk we can all go back to our comfortable world where things can be infinite, and types and sets are definitely not the same.

1. “type” = “set”
2. everything is finite*

1. "type" = "set"
2. everything is finite*

Rule number 1: types and sets are the same thing. I am going to use these words interchangeably. Rule number 2: everything is finite. OK? After my talk we can all go back to our comfortable world where things can be infinite, and types and sets are definitely not the same.

1. “type” = “set”
2. everything is finite*

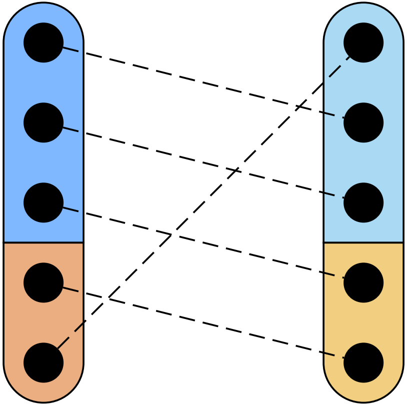
* except for that one infinite thing

Rule number 1: types and sets are the same thing. I am going to use these words interchangeably. Rule number 2: everything is finite. OK? After my talk we can all go back to our comfortable world where things can be infinite, and types and sets are definitely not the same.

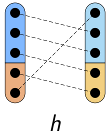
Subtraction

Subtraction

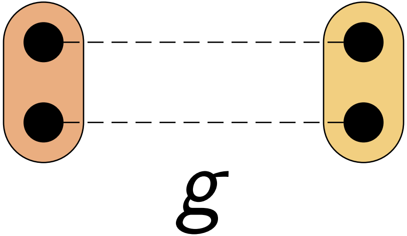
Now let's talk about subtraction.

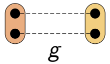


h

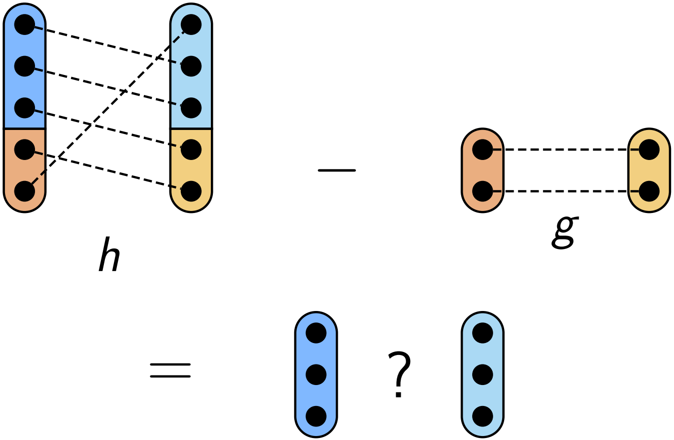


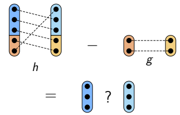
OK. Now, suppose we start with a bijection between two sum types. So here is a bijection h from, say, $a + b$, to $a' + b'$. Notice that h does not send every element in the top left to the top right, nor bottom left to bottom right. It can arbitrarily "mix" top and bottom. Put another way, h is not the sum of two bijections on the blue and orange sets.


g



Now let's take our same g again.


h
−
g
=
?



Since we can add two bijections, the natural question is—can we subtract them as well? Now at this point it may not even be clear what this should mean, especially since we just said h is not a sum of bijections. One thing we can say for sure is that the blue sets must have the same size, since h shows that the disjoint unions have the same size, and g shows that the orange sets have the same size. So there must exist some bijection between the blue sets. But this isn't good enough for me. I don't just want to know they have the same size, I want a concrete matching between the blue sets that I can actually compute.

Background

Background

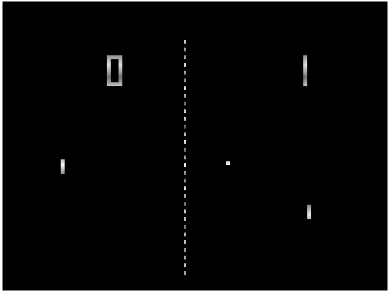
Before I explain the answer, I want to stop to give a bit of context.

Garsia-Milne (1981), Gordon (1983)

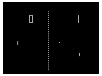
Garsia-Milne (1981), Gordon (1983)

The problem was first solved by Garsia and Milne, and later in a different form by Gordon. Both actually proved much more general things than what we will talk about here; ask me later if you're interested.

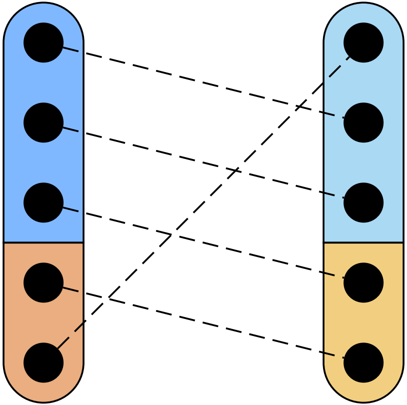
Ping-pong



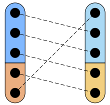
Ping-pong



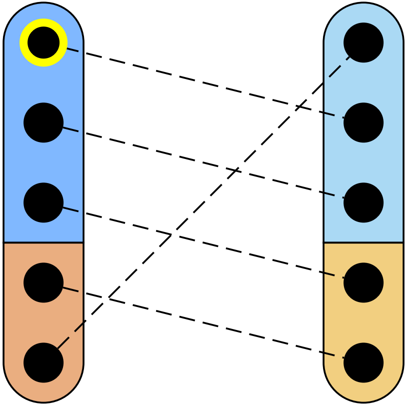
So now I want to explain the solution.



h



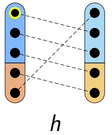
Let's start by looking at h again.



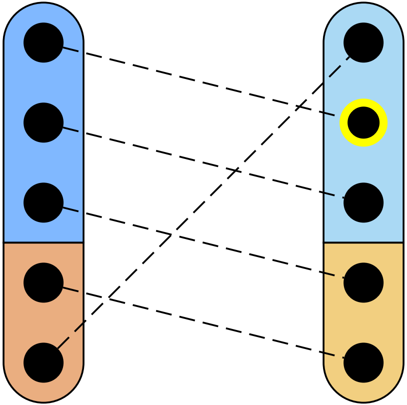
h

2018-10-20

What's the Difference? A Functional Pearl on Subtracting Bijections

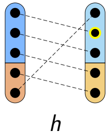


If we start with this element and follow h across. . .

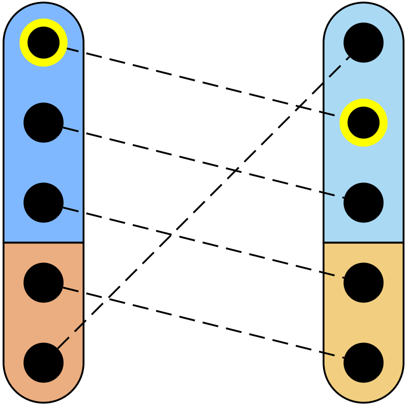


h

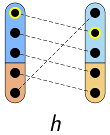
What's the Difference? A Functional Pearl on Subtracting Bijections



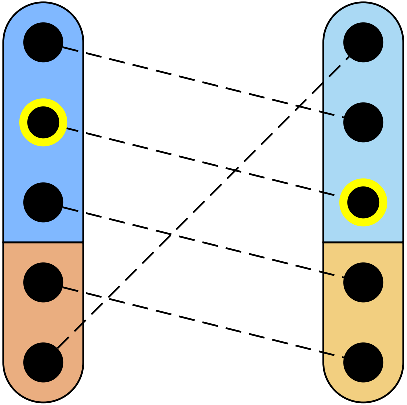
. . . we end up here. This is where we want to be—remember, we're trying to match up the blue sets.



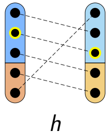
h



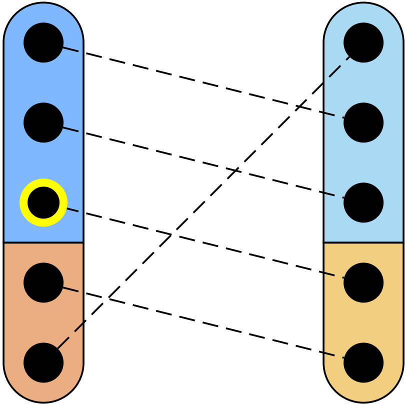
So we decide to match up these two elements.



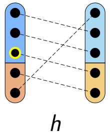
h



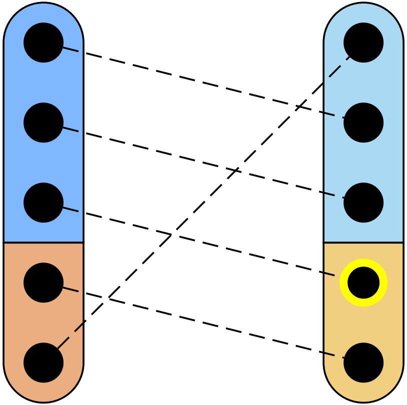
Similarly, we can match these two as well.



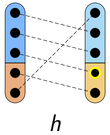
h



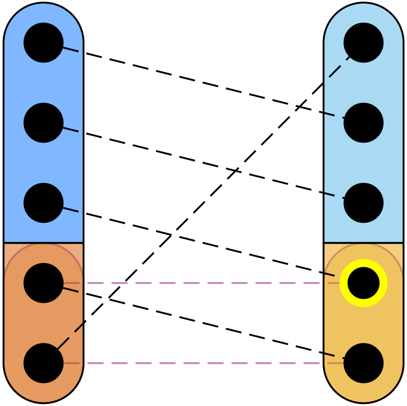
What about this one? Of course there's only one element left we can pair it with, but let's see if we can figure out a principled reason to choose it.



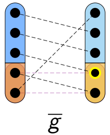
h



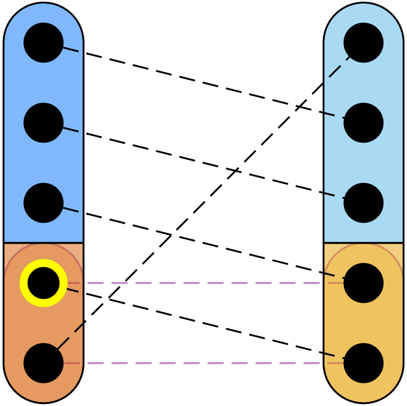
When we follow h across, we end up in the "wrong" set. What do we do from here?



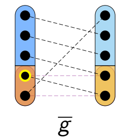
$$\overline{g}$$



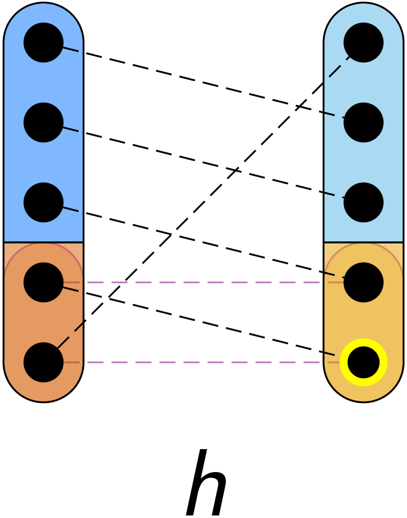
Well, remember that we have another bijection, g, which connects the orange sets. Let's superimpose it here. I've written $\overline{g}$, denoting the inverse of g, to emphasize that (as you may have already figured out) we're going to follow it backwards.

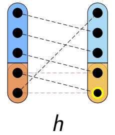


$$\overline{g}$$

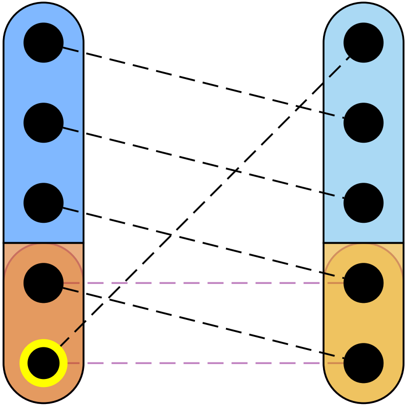


So we follow g backwards and of course we end up in the dark orange set.

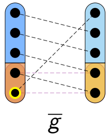

h



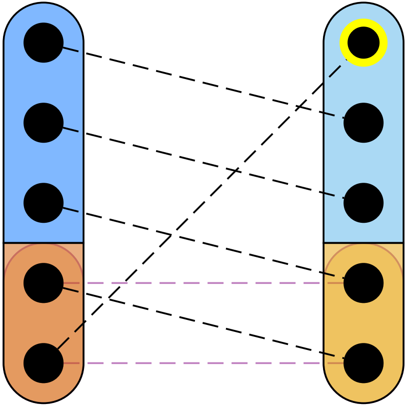
But now we can follow h again, to over here. This still isn't where we want to be...



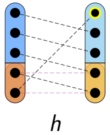
$$\overline{g}$$



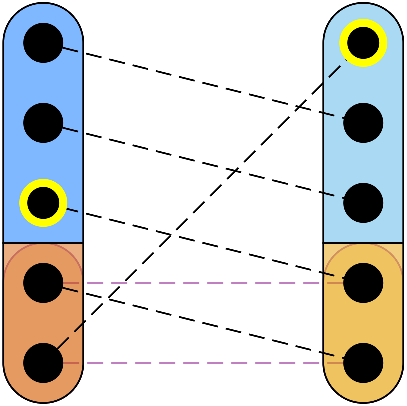
…so we follow g backwards again, to here…



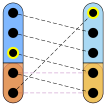
h



Then we follow h again, and finally we end up in the light blue set!

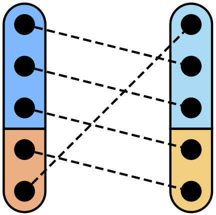


h



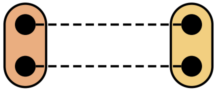
h

So we do in fact match up these elements. And we got there by sort of "ping-ponging" back and forth between the two sides, alternately following h and $\overline{g}$.



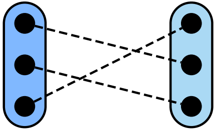
h

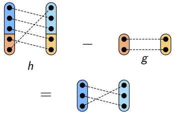
$-$



g

$=$





Overall, then, this is the bijection we get when we subtract g from h. Since everything is a bijection, and the sets are finite, we can't keep ping-ponging forever, we can't get stuck, and two different elements on the left can never end up mapping to the same element on the right.

OK, so let's see some code!

$$
\begin{aligned}
&\mathit{pingpong} :: (\mathit{Either}\ a\ b \to \mathit{Either}\ a'\ b') \to (b' \to b) \to (a \to a') \\
&\mathit{pingpong}\ h\ g\text{'} = \mathit{untilLeft}\ (h \circ \mathit{Right} \circ g\text{'}) \circ h \circ \mathit{Left}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{untilLeft} :: (b' \to a' + b') \to (a' + b' \to a') \\
&\mathit{untilLeft}\ \mathit{step}\ \mathit{ab} = \textbf{case}\ \mathit{ab}\ \textbf{of} \\
&\quad \mathit{Left}\quad a' \to a' \\
&\quad \mathit{Right}\ b' \to \mathit{untilLeft}\ \mathit{step}\ (\mathit{step}\ b')
\end{aligned}
$$

```
pingpong :: (Either a b → Either a′ b′) → (b′ → b) → (a → a′)
pingpong h g′ = untilLeft (h ∘ Right ∘ g′) ∘ h ∘ Left

untilLeft :: (b′ → a′ + b′) → (a′ + b′ → a′)
untilLeft step ab = case ab of
  Left  a′ → a′
  Right b′ → untilLeft step (step b′)
```

. . . yuck, right? This is just about the prettiest I can make it. There are a lot of problems here. There's a lot of noise injecting into and projecting from sum types. We're following individual elements rather than building bijections at a high level. And this is only one direction of the bijection! We would need to basically duplicate this code to handle the other direction.

$$
\begin{aligned}
&\mathit{pingpong} :: (\mathit{Either}\ a\ b \to \mathit{Either}\ a'\ b') \to (b' \to b) \to (a \to a') \\
&\mathit{pingpong}\ h\ g' = \mathit{untilLeft}\ (h \circ \mathit{Right} \circ g') \circ h \circ \mathit{Left}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{untilLeft} :: (b' \to a' + b') \to (a' + b' \to a') \\
&\mathit{untilLeft}\ \mathit{step}\ \mathit{ab} = \mathbf{case}\ \mathit{ab}\ \mathbf{of} \\
&\quad \mathit{Left}\ \ \ a' \to a' \\
&\quad \mathit{Right}\ b' \to \mathit{untilLeft}\ \mathit{step}\ (\mathit{step}\ b')
\end{aligned}
$$

```
pingpong :: (Either a b → Either a′ b′) → (b′ → b) → (a → a′)
pingpong h g′ = untilLeft (h ∘ Right ∘ g′) ∘ h ∘ Left

untilLeft :: (b′ → a′ + b′) → (a′ + b′ → a′)
untilLeft step ab = case ab of
  Left  a′ → a′
  Right b′ → untilLeft step (step b′)
```

…yuck, right? This is just about the prettiest I can make it. There are a lot of problems here. There’s a lot of noise injecting into and projecting from sum types. We’re following individual elements rather than building bijections at a high level. And this is only one direction of the bijection! We would need to basically duplicate this code to handle the other direction.

So let's get rid of that ugly code. Ah, much better! So, Kenny and I set out to see if we could find a way to construct this algorithm in a high-level, point-free way. Why? Partly just as a fun challenge, and also to gain insight into the algorithm and the related combinatorics. We also hoped it could be a first step towards building a formal computer proof.

Guðmundsson (2017)

What's the Difference? A Functional Pearl on Subtracting Bijections

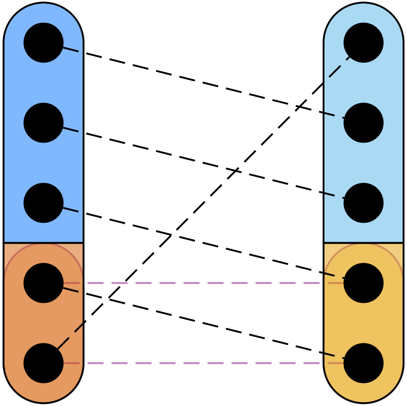
Guðmundsson (2017)

At the time we started working on this, there were no formal computer proofs that we knew of; last year Guðmundsson completed a formal proof in Agda for his master's thesis, though it is pretty tedious, and low-level; turning our approach into a higher-level formal proof is future work.

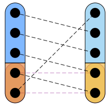
High-level ping-pong

High-level ping-pong

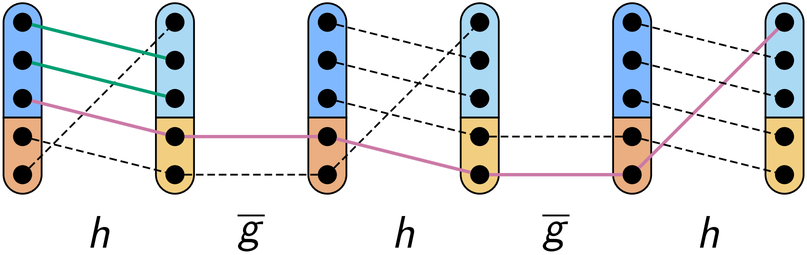
So let's play some high-level ping-pong.

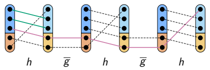


$h, \overline{g}$



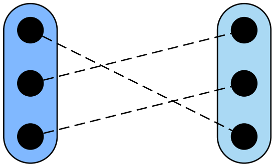
Our first step is to unfold the ping-ponging process. Instead of thinking of h and g being superimposed and watching elements bounce back and forth...

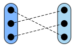

h
$\overline{g}$
h
$\overline{g}$
h



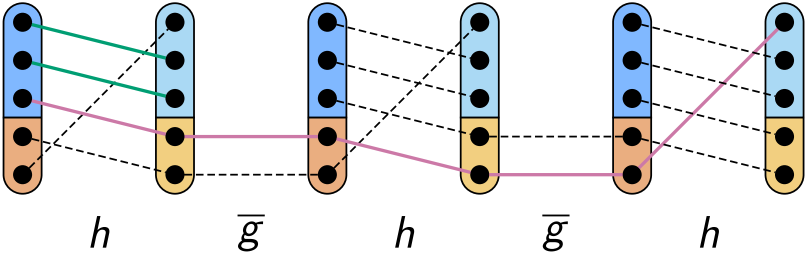
…we can visualize time using a spatial dimension, and unfold the process into a sort of "trace" through multiple copies of the sets. I have highlighted the paths taken by each of the three elements.

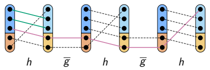
Not only is this a nicer way to visualize the process, but it gives us an idea. This trace is built out of a bunch of bijections glued together. Maybe we can build an entire trace in a high-level, compositional way, and then extract the bijection we want at the end.



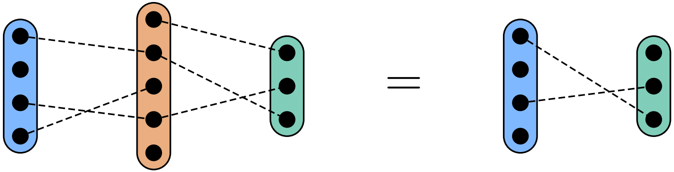


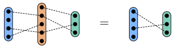
So what is a bijection? We can represent a bijection between types a and b simply as a pair of functions from $a \to b$ and $b \to a$; of course we also require that the two functions compose to the identity. There is an id bijection and we can compose them, that is, they form a category.


h
$\overline{g}$
h
$\overline{g}$
h



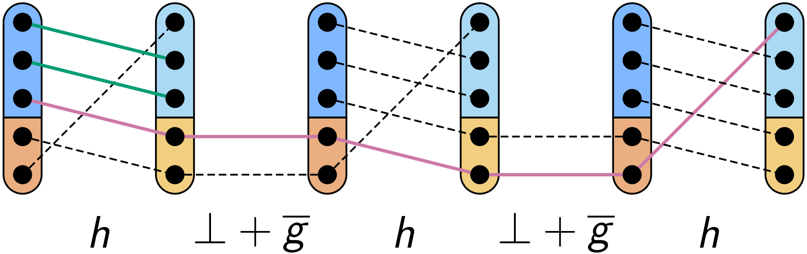
Going back to this for a minute, we can see that bijections aren't enough. . . notice these gaps. The types don't match up, since g is only defined on the orange sets. So we introduce the notion of partial bijections.

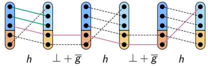




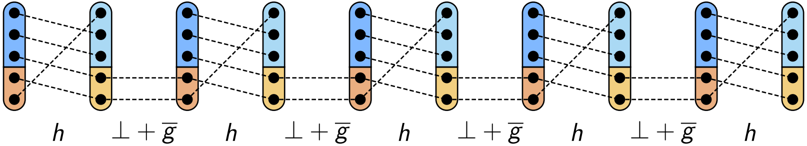
It turns out that bijections aren't enough. We also need partial bijections, which are like bijections except that they may be undefined in some places.

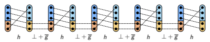
Formally, we can define a partial bijection as a pair of partial functions in opposite directions. We can do all the same things with them as with total bijections, like compose them in sequence and in parallel. The composition works like...


h
$\perp + \overline{g}$
h
$\perp + \overline{g}$
h

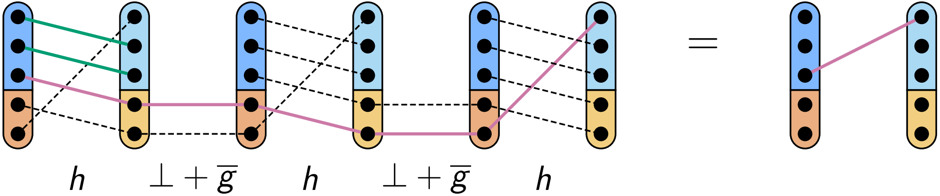


So now we can finally put the pieces together to construct a trace. We compose the empty partial bijection in parallel with the inverse of g for the intermediate steps; then we compose an alternating sequence of this with h. Incidentally, I will use semicolon to indicate "backwards" composition, so values flow from left to right, in the same direction as the diagrams.

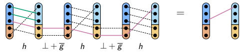

h
$\perp + \overline{g}$
h
$\perp + \overline{g}$
h
$\perp + \overline{g}$
h
$\perp + \overline{g}$
h



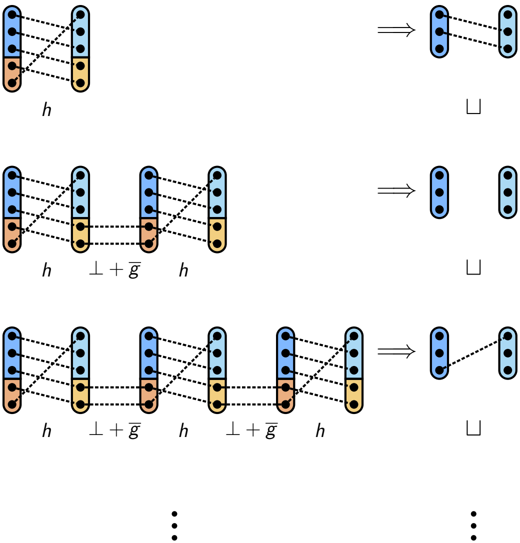
Unfortunately, this doesn't actually work! First, how do we know how many times to iterate?



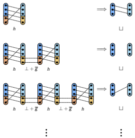
h
$\perp + \overline{g}$
h
$\perp + \overline{g}$
h
$=$



And even if we did know how many times to iterate, it still doesn't work: the actual result of composing this trace is a partial bijection containing only the purple path. The problem is that the other paths stop too early, so they get lost. Remember that an edge will show up in the final composed output only if there is a complete, unbroken path all the way from one side to the other!



h
$\sqcup$
h
$\perp + \overline{g}$
h
$\sqcup$
h
$\perp + \overline{g}$
h
$\perp + \overline{g}$
h
$\sqcup$



These are all compatible (see paper), so if we take the infinite merge (as long as it is lazy enough), we get exactly what we wanted!

What's the Difference? A Functional Pearl on Subtracting Bijections

BRENT A. YORGEY, Hendrix College, USA
KENNETH FONER, University of Pennsylvania, USA

It is a straightforward exercise to write a program to "add" two bijections—resulting in a bijection between two sum types, which runs the first bijection on elements from the left summand and the second bijection on the right. It is much less obvious how to "subtract" one bijection from another. This problem has been studied in the context of combinatorics, with several computational principles known for producing the "difference" of two bijections. We consider the problem from a computational and algebraic perspective, showing how to construct such bijections at a high level, avoiding pointwise reasoning or being forced to construct the forward and backward directions separately—without sacrificing performance.

CCS Concepts: • **Mathematics of computing** → **Combinatorics**; • **Software and its engineering** → **Functional languages**;

Additional Key Words and Phrases: bijection, difference

ACM Reference Format:
Brent A. Yorgey and Kenneth Foner. 2018. What's the Difference? A Functional Pearl on Subtracting Bijections. *Proc. ACM Program. Lang.* 2, ICFP, Article 101 (September 2018), 21 pages. https://doi.org/10.1145/3236796

1 INTRODUCTION

Suppose we have four finite types (sets) A, B, A', and B' with bijections $f : A \leftrightarrow A'$ and $g : B \leftrightarrow B'$. Then, as illustrated[1] in Figure 1, we can "add" these bijections to produce a new bijection

$$h : A + B \leftrightarrow A' + B',$$

where + denotes a sum type (or a disjoint union of sets). We take h to be the function which applies f on elements of A, and g on elements of B, which we denote as $h = f + g$. In Haskell, we could encode this as follows:

type $(+) = \mathit{Either}$

$(+) :: (a \to a') \to (b \to b') \to (a + b \to a' + b')$
$(f + g)\ (\mathit{Left}\ x) \quad = \mathit{Left} \quad (f\ x)$
$(f + g)\ (\mathit{Right}\ y) = \mathit{Right}\ (g\ y)$

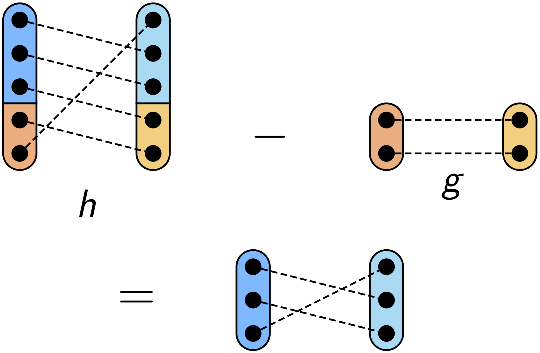
(Note we are punning on (+) at the value and type levels. This function already lives in the standard `Data.Bifunctor` module with the name *bimap*—in the *Bifunctor Either* instance—but for our

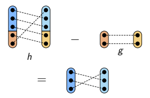
[1] We recommend viewing this paper as a PDF or printing it on a color printer, though it should still be comprehensible in black and white. The colors have been chosen to remain distinguishable to individuals with common forms of color blindness.

Authors' addresses: Brent A. Yorgey, Department of Mathematics and Computer Science, Hendrix College, Conway, AR, USA, yorgey@hendrix.edu; Kenneth Foner, University of Pennsylvania, Philadelphia, PA, USA, kfoner@seas.upenn.edu.

2475-1421/2018/9-ART101
https://doi.org/10.1145/3236796

There's a bunch more in the paper. For example, this infinite merge solution works but suffers from quadratic performance for two different reasons, and we show how to make the performance linear again without too much modification to the code.


h
−
g
=



So, thanks very much for listening, and go read the paper!