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.* 1, ICFP, Article 1 (January 2018), 20 pages.

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

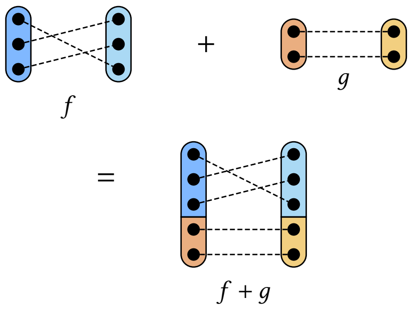


Fig. 1. Adding bijections

[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.

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 purposes it is clearer to just define our own.) We can see that $(f + g)$ is a bijection as long as f and g are.

So we can define the *sum* of two bijections. What about the *difference*? That is, given bijections h and g with

$$\begin{aligned} h &: A + B \leftrightarrow A' + B' \\ g &: \quad\;\;\, B \leftrightarrow \quad\;\;\, B', \end{aligned}$$

can we compute some

$$f : A \quad \leftrightarrow A' \quad ?$$

Certainly A and A' must have the same size. The existence of the bijections h and g tells us that $|A + B| = |A' + B'|$ and $|B| = |B'|$; since $|A + B| = |A| + |B|$, and similarly for $|A' + B'|$ (keeping in mind that $+$ is *disjoint* union), we can just subtract the second equation from the first to conclude that $|A| = |A'|$. Since A and A' are finite sets with the same size, there *must exist* some bijection $A \leftrightarrow A'$. But this is not constructive: what if we want to actually *compute* a bijection $A \leftrightarrow A'$? The fact that A and A' have the same size, in and of itself, does not help us actually match up their elements. The goal is to somehow use the *computational content* of the bijections h and g to come up with a (suitably canonical) definition for $h - g$.

To see why this problem is not as trivial as it may first seem, consider Figure 2. The problem

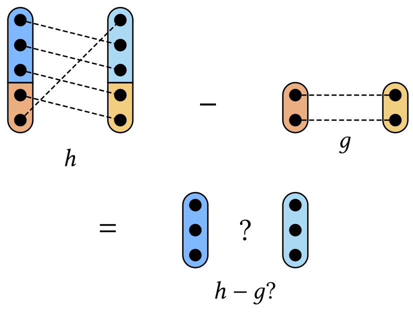


Fig. 2. Subtracting bijections?

is that the bijection $h : A + B \leftrightarrow A' + B'$ may not look like a sum of bijections $f + g$. A sum of bijections always keeps A and B separate, mapping A into A' and B into B' (as in Figure 1). However, in general, a bijection $h : A + B \leftrightarrow A' + B'$ may arbitrarily mix elements between the sets. In Figure 2, notice how h sends some elements from A (dark blue) to B' (light orange) and likewise from B (dark orange) to A' (light blue). In general, then, we cannot do anything so simple as just "drop" B and B'. We will somehow need to make use of g as well.

Before attacking this problem, it's worth taking a minute to consider why it is worth studying. This problem was first studied (and solved) in the context of combinatorics, where proving merely that two sets must have the same size is usually considered unsatisfactory: the goal is to exhibit an explicit bijection that serves as a (constructive) witness of the fact, and helps uncover additional structure or give some intuition as to "why" the sets have the same size. Relatedly, in order to use techniques from combinatorics in the context of a proof assistant based on constructive logic, being able to subtract bijections constructively is important.

In one sense, the problem has already been solved, first by Garsia and Milne [1981] and then, in a different form, by Gordon [1983]. However, the usual presentation of their techniques is low-level and element-based (*i.e.* "pointful"), which obscures the high-level details; in addition, since the construction is usually presented in one direction at a time, there is an extra burden of proof to show that the two directions so constructed actually form a bijection.

Using an algebra of *partial bijections*, we will give a high-level construction of the Gordon complementary bijection principle (GCBP), which computes the difference of two bijections. One downside of our high-level implementation of GCBP is that one direction of the computed bijection has quadratic performance, whereas the usual low-level implementation takes linear time in both directions. However, we will then show how to optimize the implementation so that both directions run in linear time again, while retaining its high-level character.

2 THE GORDON COMPLEMENTARY BIJECTION PRINCIPLE

Let us return to the problem of computing some $h - g : A \leftrightarrow A'$ from $h : A + B \leftrightarrow A' + B'$ and $g : B \leftrightarrow B'$ and describe the solution of Gordon [1983] as it is typically presented. The key to defining $h - g$ is to use h and g to "ping-pong" back and forth between sets until landing in the right place.

Starting with an arbitrary element of A, our goal is to find an element of A' to match it with. First, run it through $h : A + B \leftrightarrow A' + B'$. If we land in A', we are done. Otherwise, we end up with an element of B'. Run this backwards through $g : B \leftrightarrow B'$, yielding an element of B. Now run h again. This may yield an element of A', in which case we can stop, or an element of B'; we continue iterating this process until finally landing in A'. We then match the original element of A to the element of A' so obtained.

Figure 3 illustrates this process. The top two elements of the (dark blue) set on the upper-left map immediately into the two lower elements of the light blue set; the third element of the dark blue set, however, requires two iterations back and forth before finally landing on the uppermost element of the light blue set. Symbolically, for each $a \in A$ we find the smallest n such that $(h\overline{g})^n h$

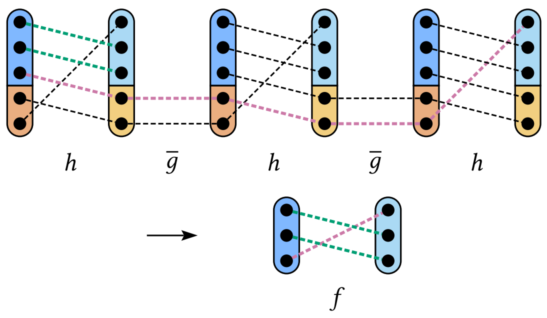


Fig. 3. Ping-ponging

applied to a yields some $a' \in A'$. (We denote the inverse of a bijection $f : X \leftrightarrow Y$ by $\overline{f} : Y \leftrightarrow X$, and denote the composition of bijections by juxtaposition, that is, $fg(a) = (f \circ g)(a) = f(g(a))$.) Figure 4 contains a basic Haskell implementation of this process. It is worth pointing out that the Haskell implementation is a bit noisier because of the need for *Left* and *Right* constructors; typical mathematical presentations treat A as a subset (subtype) of $A + B$, so that an element $a \in A$ *is also* an element of $A + B$, without the need to explicitly inject it.

```
pingpong :: (a + b → 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′)
```

Fig. 4. Ping-ponging in Haskell

Gordon [1983] introduced this algorithm, and called it the *complementary bijection principle*. (Actually, Gordon's principle is quite a bit more general than this, involving a whole tower of bijections on chains of nested subsets, but we do not consider the principle in full generality here.)

At this point, it's worth going through a careful, standard proof of the complementary bijection principle. We must prove two things: first, that the algorithm terminates; second, that it actually produces a bijection, as claimed.

Proof. We first prove that the algorithm terminates. Let $a \in A$ and consider the sequence of values in $A' + B'$ generated by the algorithm:

$$h(a), \quad h\overline{g}h(a), \quad h\overline{g}h\overline{g}h(a), \quad \ldots,$$

which we can write more compactly as

$$((h\overline{g})^n h)(a)$$

for $n \geqslant 0$. The claim is that $((h\overline{g})^n h)(a) \in A'$ for some n, at which point the algorithm stops. Suppose not, that is, suppose $((h\overline{g})^n h)(a) \in B'$ for all $n \geqslant 0$. Then since B' is finite, by the pigeonhole principle there must exist $0 \leqslant j < k$ such that

$$((h\overline{g})^j h)(a) = ((h\overline{g})^k h)(a).$$

Since h and g are bijections, so is $(h\overline{g})^j$, and we may apply its inverse to both sides, obtaining

$$h(a) = ((h\overline{g})^{k-j} h)(a) = (h(\overline{g}h)^{k-j})(a).$$

Finally, applying $\overline{h}$ to both sides,

$$a = (\overline{g}h)^{k-j}(a).$$

But $a \in A$, whereas the right-hand side is an element of the codomain of $\overline{g}$, that is, B. This is a contradiction, so the algorithm must in fact terminate for any starting $a \in A$.

It remains to show that the function f so constructed is a bijection. Note that given a particular $a \in A$, the algorithm visits a series of elements in $A, B', B, B', \ldots, B, A'$, finally stopping when it reaches A', and assigning the resulting element of A' as the output of $f(a)$. We can explicitly construct $\overline{f}$ by running the same algorithm with $\overline{h}$ and $\overline{g}$ as input in place of h and g. That is, intuitively, we build Figure 3 from right to left instead of left to right. When run "in reverse" in this way on $f(a)$, we can see that the algorithm will visit exactly the same series of elements in reverse, stopping when it reaches the original a since it will be the first element not in B. □

This construction and proof would convince any combinatorialist, but they have several downsides:

- This presentation makes heavy use of "pointwise" reasoning, messily following the fate of individual elements through an algorithm. We would like a "higher-level" perspective on both the algorithm and proof. Note we cannot simply rewrite the above algorithm in terms of function composition and get rid of any mentions of a, since the algorithm may iterate a different number of times for each particular $a \in A$.
- Relatedly, in this presentation we construct the forward and backward directions separately, and then prove that the results are inverse. It would be better to construct both directions of the bijection simultaneously, so that the resulting bijection is "correct by construction".

We solve these problems by eschewing point-based reasoning in favor of a high-level algebraic approach, which we use to directly construct a bijection which is the "difference" of two other bijections. We also hope that a high-level construction may be easier to formally prove correct. Guðmundsson [2017] has recently given the first mechanized formal proof of the complementary bijection principle, in Agda, but it is long and relies heavily on low-level pointwise reasoning. We leave to future work the challenge of turning our high-level construction into a corresponding high-level proof.

3 PARTIAL BIJECTIONS

Since the GCBP takes two bijections as input and yields a bijection as output, one might think to begin by defining a type of bijections:

```
data Bijection a b = Bijection
  { fwd :: a → b
  , bwd :: b → a
  }
```

satisfying the invariants that $to \circ from = id$ and $from \circ to = id$. (In a dependently typed language, one might well include these conditions as part of the definition. Although Haskell itself can simulate dependent types, invariants such as these would make our definitions much too heavyweight.) The idea would be to somehow piece together the GCBP algorithm out of high-level operations on bijections, so that the whole algorithm returns a valid bijection by construction, eliminating duplication of code and the possibility for the forward and backward directions to be out of sync.

This is the right idea, but it is not good enough. The problem is that when it comes to bijections, the algorithm is an all-or-nothing sort of deal: we put two bijections in and get one out, but it is hard to find intermediate bijections that arise during execution of the algorithm, out of which we could build the ultimate result.

Instead, we must generalize to *partial* bijections, that is, bijections which may be undefined on some parts of their domain (Figure 5). We can think of the algorithm as starting with a totally undefined bijection and building up more and more information, until finishing with a total bijection.

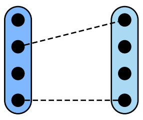


Fig. 5. A partial bijection

Whereas a (total) bijection consists of a pair of inverse functions $a \rightarrow b$ and $b \rightarrow a$, a partial bijection consists of a pair of *partial* functions $a \rightarrow \mathit{Maybe}\ b$ and $b \rightarrow \mathit{Maybe}\ a$ which are "inverse" in a suitable sense (to be precisely defined later). We can work uniformly with both by generalizing to an arbitrary *Kleisli* category,

```
newtype $a \rightarrow_m b = K\ \{\mathit{runKleisli} :: a \rightarrow m\ b\}$
```

consisting of functions $a \rightarrow m\ b$ for any monad m. In one sense, this generality is overkill: working concretely with total and partial bijections instead of a common generalization would require a bit of code duplication but would be quite a bit simpler. However, in addition to reducing code duplication, working in a more general setting reveals some underlying algebraic structure, and hints at potential extensions and generalizations to be discovered.

In any case, picking $m = \mathit{Identity}$ in the definition of $a \rightarrow_m b$ yields normal total functions (up to some extra **newtype** wrappers); picking $m = \mathit{Maybe}$ yields partial functions. The *Category* instance for $\cdot \rightarrow_m \cdot$ provides the identity $\mathit{id} :: a \rightarrow_m a$ along with a composition operator

$$(\circ) :: (b \rightarrow_m c) \rightarrow (a \rightarrow_m b) \rightarrow (a \rightarrow_m c).$$

```
class $\mathit{Category}\ \mathit{cat}$ where
  $\mathit{id}\ \ :: \mathit{cat}\ a\ a$
  $(\circ) :: \mathit{cat}\ b\ c \rightarrow \mathit{cat}\ a\ b \rightarrow \mathit{cat}\ a\ b$
instance $\mathit{Monad}\ m \Rightarrow \mathit{Category}\ (\cdot \rightarrow_m \cdot)$ where
  $\mathit{id}\qquad\quad = K\ \mathit{return}$
  $K\ g \circ K\ f = K\ (\lambda a \rightarrow f\ a \gg\!= g)$
```

In order to match up with the pictures, where we tend to draw functions going from left to right, we will also make use of the notation

```
$f\ ;g = g \circ f$
```

We can now define a general type of m-bijections as

```
data $a \leftrightsquigarrow_m b = B$
  { $\mathit{fwd} :: a \rightarrow_m b$
  , $\mathit{bwd} :: b \rightarrow_m a$
  }
```

These compose via a *Category* instance, and also have inverses:

```
instance $\mathit{Monad}\ m \Rightarrow \mathit{Category}\ (\cdot \leftrightsquigarrow_m \cdot)$ where
  $\mathit{id} = B\ \mathit{id}\ \mathit{id}$
  $(B\ f_1\ g_1) \circ (B\ f_2\ g_2) = B\ (f_1 \circ f_2)\ (g_2 \circ g_1)$
class $\mathit{Category}\ c \Rightarrow \mathit{Groupoid}\ c$ where
  $(\overline{\cdot}) :: c\ a\ b \rightarrow c\ b\ a$
instance $\mathit{Monad}\ m \Rightarrow \mathit{Groupoid}\ (\cdot \leftrightsquigarrow_m \cdot)$ where
  $\overline{B\ f\ g} = B\ g\ f$
```

Composing two partial functions works the way we would expect: the composite $f\ ; g$ is defined on a if and only if f is defined on a and g is defined on $f(a)$. Thinking pictorially, when we compose two partial bijections, we keep only those paths which travel continuously across the entire diagram, as shown in Figure 6.

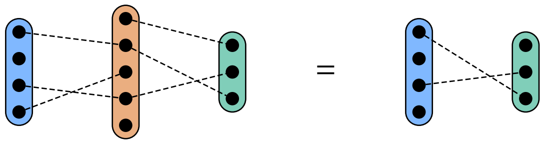


Fig. 6. Composing partial bijections

$$\textbf{pattern}\ (:\rightleftharpoons:)\ f\ g = B\ (K\ f)\ (K\ g)$$

$$\begin{aligned}
&\textbf{pattern}\ (:\leftrightarrow:)\ f\ g \leftarrow B\ (K\ ((;runIdentity) \rightarrow f))\\
&\qquad\qquad\qquad\qquad\quad (K\ ((;runIdentity) \rightarrow g))\\
&\quad \textbf{where}\\
&\qquad f :\leftrightarrow: g = B\ (K\ (f\ ;Identity))\\
&\qquad\qquad\qquad\quad (K\ (g\ ;Identity))
\end{aligned}$$

Fig. 7. Pattern synonyms for total and partial bijections

Not just any pair of Kleisli arrows qualifies as a generalized bijection. When $m = Identity$, a generalized bijection should consist of two inverse functions, that is, functions whose composition is *id*. When $m = Maybe$, composing the two functions does not have to yield the identity, since it may be undefined in some places—but it should certainly be the identity when restricted to points on which the functions are defined. More formally, we should have $fwd\ a = Just\ b$ if and only if $bwd\ b = Just\ a$. This justifies drawing partial bijections by connecting two sets with some collection of undirected (*i.e.* bidirectional) line segments, as in Figure 5. (These laws can be generalized to any monad m with some sort of "information ordering" relation; intuitively the composition of the *fwd* and *bwd* morphisms should be the identity "up to any m effects".)

As an aside, we remark that choosing $m = Set$ (which is a monad in the mathematical sense, if not a *Monad* in Haskell) leads to "multibijections", where each element $a \in A$ may map to zero or more elements in B, as long as each such element in the image of a also maps back to a (and possibly some other elements of A). We leave to future work the question of whether there is any interesting analogue to the bijection principles discussed here for multibijections.

Finally, we can recover specific types for total and partial bijections as

$$\begin{aligned}
&\textbf{type}\ a \leftrightarrow b = a \leftrightsquigarrow_{Identity} b\\
&\textbf{type}\ a \rightleftharpoons b = a \leftrightsquigarrow_{Maybe} b
\end{aligned}$$

To make working with total and partial bijections more convenient, we can define *pattern synonyms* [Pickering et al. 2016] which let us pretend as if we had directly declared types like

$$\begin{aligned}
&\textbf{data}\ a \leftrightarrow b = (a \rightarrow b) &&:\leftrightarrow: (b \rightarrow a)\\
&\textbf{data}\ a \rightleftharpoons b = (a \rightarrow Maybe\ b) &&:\rightleftharpoons: (b \rightarrow Maybe\ a)
\end{aligned}$$

automatically handling the required **newtype** wrapping and unwrapping. The declarations for these pattern synonyms are shown in Figure 7 for completeness, though the syntax is somewhat complex and the details are unimportant.

In what follows, we will sometimes use simple diagrams of labelled boxes and lines to abstractly represent sets and generalized bijections between them, since looking at the pictures gives a much better intuitive idea of what is going on than looking at code. For example, we draw a generalized

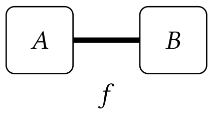


Fig. 8. A generalized bijection f between A and B

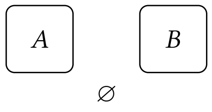


Fig. 9. The undefined partial bijection

$$\begin{aligned}
&\mathit{applyTotal} :: (a \leftrightarrow b) \longrightarrow a \longrightarrow b\\
&\mathit{applyTotal}\ (f :\!\leftrightarrow\!: \_) = f\\
&\mathit{applyPartial} :: (a \rightleftharpoons b) \longrightarrow a \longrightarrow \mathit{Maybe}\ b\\
&\mathit{applyPartial}\ (f :\!\rightleftharpoons\!: \_) = f\\
&\varnothing :: a \rightleftharpoons b\\
&\varnothing = \mathit{const}\ \mathit{Nothing} :\!\rightleftharpoons\!: \mathit{const}\ \mathit{Nothing}\\
&\mathit{partial} :: (a \leftrightarrow b) \longrightarrow (a \rightleftharpoons b)\\
&\mathit{partial}\ (f :\!\leftrightarrow\!: g) = (f\ ;\mathit{Just}) :\!\rightleftharpoons\!: (g\ ;\mathit{Just})\\
&\mathit{unsafeTotal} :: (a \rightleftharpoons b) \longrightarrow (a \leftrightarrow b)\\
&\mathit{unsafeTotal}\ (f :\!\rightleftharpoons\!: g) = (f\ ;\mathit{fromJust}) :\!\leftrightarrow\!: (g\ ;\mathit{fromJust})
\end{aligned}$$

Fig. 10. Utilities for partial and total bijections

bijection f between sets A and B as a thick line connecting two labelled boxes, as shown in Figure 8. Note that we don't draw the action of f on individual elements of A and B, but simply summarize the relationship expressed by f with a single thick line.

We now define some utility functions for working with total and partial bijections (Figure 10). First, *applyTotal* and *applyPartial* let us run a bijection in the forward direction. Next, we define $\varnothing$ as the totally undefined partial bijection, which we draw as two unconnected boxes, as in Figure 9. Finally, the *partial* and *unsafeTotal* functions move back and forth between total and partial bijections. Treating a total bijection as a partial one is always safe, and we will sometimes omit calls to *partial* in informal discussion, thinking of total bijections as a "subtype" of partial bijections. On the other hand, moving in the other direction is only safe if we know that the "partial" bijection is, in fact, defined everywhere. Evaluating *unsafeTotal* f at some input for which f is undefined will result in a runtime error. (With a richer type system we could of course make a safe version of *unsafeTotal* which requires a proof of totality.)

We now turn to developing tools for dealing with bijections involving sum types. It is useful to have a type class for "things which compose in parallel". If f is some sort of relation between A and A', and g relates B and B', then their parallel sum $f + g$ relates the disjoint sums $A + B$ and $A' + B'$, which we visualize by stacking vertically (Figure 11). For example, normal functions $A \longrightarrow A'$ compose in parallel: if $f : A \longrightarrow A'$ and $g : B \longrightarrow B'$ then $f + g : A + B \longrightarrow A' + B'$ is the function which runs f on elements of A and g on elements of B; we saw this parallel composition of functions in

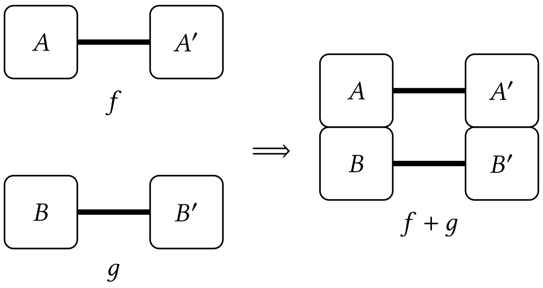


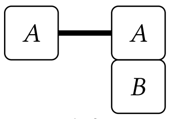
Fig. 11. Parallel composition (sum) of generalized bijections

```
class Category arr ⇒ Parallel arr where
  (+) :: arr a c → arr b d → arr (a + b) (c + d)
instance Parallel (→) where
  (+) = bimap
factor :: Functor m ⇒ m a + m b → m (a + b)
factor = either (fmap Left) (fmap Right)
instance Monad m ⇒ Parallel (· →_m ·) where
  K f + K g = K ((f + g) ; factor)
instance Monad m ⇒ Parallel (· ⇝_m ·) where
  (B f g) + (B h i) = B (f + h) (g + i)
```

Fig. 12. Parallel composition

the introduction. Kleisli arrows over the same monad m also compose in parallel: the parallel sum of $f : A \rightarrow_m A'$ and $g : B \rightarrow_m B'$ works the same as the parallel sum of normal functions, but has the effects of whichever one actually runs. Finally, since Kleisli arrows compose in parallel, so can generalized bijections, by composing the forward and backward directions separately. The code is shown in Figure 12.

We also define *left*, the partial bijection which injects A into $A + B$ in one direction, and drops B in the other direction:



left

From this we define the *left partial projection.* If $f : A + B \rightleftharpoons A' + B'$, then the left partial projection of f, denoted $\langle f \,|$, has the type $\langle f \,| : A \rightleftharpoons A'$, and consists of only those edges of f which go between A and A'. Any edges involving elements in B or B' are dropped. This can be accomplished simply by sandwiching f in between *left* and $\overline{\textit{left}}$, as shown in Figure 13; code for *left* and $\langle \cdot \,|$ is shown in Figure 14. (The functions *right* and $|\cdot\rangle$ could be defined similarly, but we do not need them.) Figure 15 shows an example of computing the left partial projection $\langle h \,|$, using the same bijection h from the introduction.

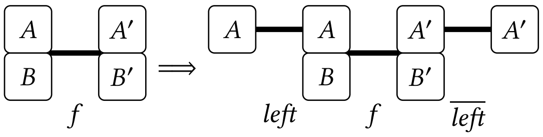


Fig. 13. Constructing the left partial projection via *left*

$$\begin{aligned}
&\mathit{left} :: a \rightleftharpoons a + b\\
&\mathit{left} = (\mathit{Left} \mathbin{;} \mathit{Just}) :\rightleftharpoons: \mathit{either}\ \mathit{Just}\ (\mathit{const}\ \mathit{Nothing})\\
&\langle \cdot \,| :: (a + b \rightleftharpoons a' + b') \to (a \rightleftharpoons a')\\
&\langle f\,| = \mathit{left} \mathbin{;} f \mathbin{;} \overline{\mathit{left}}
\end{aligned}$$

Fig. 14. *left* and $\langle \cdot \,|$

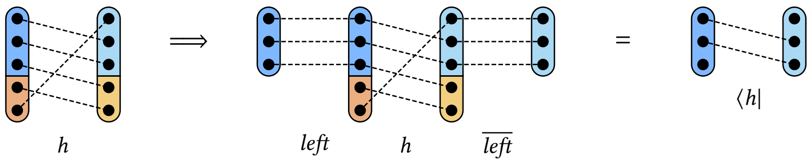


Fig. 15. Computing the left partial projection $\langle h\,|$

4 GCBP, TAKE 1

We now have the tools we need to construct a first attempt at a high-level, point-free version of the Gordon Complementary Bijection Principle. Although this first version will ultimately turn out to be unusable in practice, it has most of the important ingredients of the more sophisticated variants developed later.

The basic idea will be to construct the diagram at the top of Figure 16 step-by-step, taking h as a starting point and building up the whole diagram incrementally, accumulating more information about the final bijection as more elements land in the top set, until all elements have landed in the top set and we can stop. The left partial projection of the result (which keeps only the top half of the diagram) will then be the bijection we are looking for.

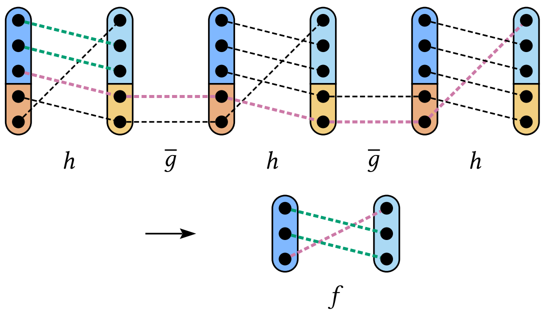


Fig. 16. The GCBP construction

Unfortunately, Figure 16 is misleading: if we literally compose copies of h and $\overline{g}$ as shown at the top of the figure and then take the left projection, we don't actually get the desired f, but

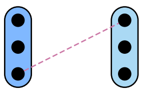


Fig. 17. The literal result of the composition in Figure 16

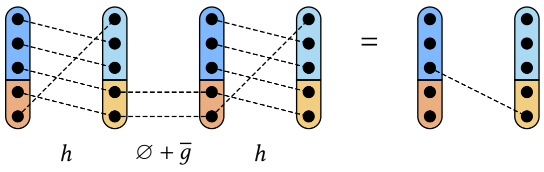


Fig. 18. The first step of GCBP

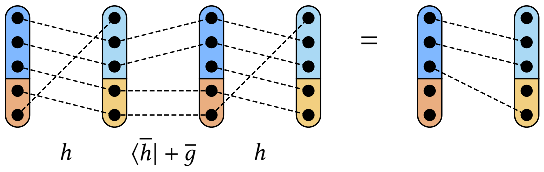


Fig. 19. The first step of GCBP, patched

rather we get the partial bijection shown in Figure 17. The problem is that some of the paths that we want to have in the final bijection actually stop short of the last iteration, so they are lost when composing the entire diagram. Only continuous paths from the top, leftmost set all the way to the top, rightmost set survive the composition; in this case, there is only one such path. The classical, elementwise formulation of GCBP specifies that the iteration for each particular element continues only until landing in the target set; the number of iterations may be different for each particular element. When building GCBP at a higher level, however, we must somehow keep track of everything at once.

Let's see how we might start building up something like Figure 16. First of all, we can't compose $h : A + B \leftrightarrow A' + B'$ with $\overline{g} : B' \leftrightarrow B$ directly, since their types do not match. (Note, once again, that in typical mathematical presentations, this is glossed over since a subtyping relationship $B' \leqslant A' + B'$ is assumed.) However, if we compose $\overline{g}$ in parallel with $\varnothing : A' \rightleftharpoons A$ we get

$$\varnothing + \overline{g} : A' + B' \rightleftharpoons A + B.$$

Sequencing h with this followed by another copy of h gives the first iteration of GCBP, shown in Figure 18. However, the problem crops up here already: the result of $h \mathbin{;} (\varnothing + \overline{g})$ is a partial bijection which is undefined on the first two elements of A (dark blue). Those elements already mapped to B (light blue) under h, so they are already "done": the only reason to keep iterating is to find out what will happen to the third element. But as soon as we start iterating we lose the knowledge of what happened to the first two.

One (bad!) idea is to "recycle" elements that have landed in B, sending them back to where they started so the cycle can repeat. That is, instead of taking the parallel sum of $\overline{g}$ with $\varnothing$ at each step, we take the parallel sum of $\overline{g}$ with the inverse of (the left partial projection of) whatever we have constructed so far. At the very first step this means using not $\varnothing + \overline{g}$ but the parallel sum of $\left\langle \overline{h} \right|$ and g, as shown in Figure 19. On the second iteration, we would take the partial bijection constructed so far, namely,

$$f_1 = h \mathbin{;} (\langle\overline{h}\,| + \overline{\mathrm{g}}) \mathbin{;} h,$$

and use its inverse to “plug the hole” over the second copy of $\overline{\mathrm{g}}$, that is,

$$f_2 = f_1 \mathbin{;} (\langle\overline{f_1}\,| + \overline{\mathrm{g}}) \mathbin{;} h,$$

and so on; in general we would define

$$f_{i+1} = f_i \mathbin{;} (\langle\overline{f_i}\,| + \overline{\mathrm{g}}) \mathbin{;} h.$$

This ensures that every time a path lands in B, it is sent back to the element in A it started from, allowing it to cycle through again while it is “waiting” for other elements to land in B.

Unfortunately, this is quite inefficient. For one thing, evaluating each f_{i+1} on a particular input requires evaluating two copies of f_i, leading to exponential time complexity to evaluate f_n at a given input (at least barring any clever optimizations). Second, there is something else we have swept under the rug up to this point: how do we know how long to keep iterating? Note that because of the way we send each path back to the start while waiting for other paths to complete, at the point when the last path completes the others may be in the middle of a cycle. So in fact, we must iterate for a number of steps equal to the least common multiple of the cycle lengths, until all the cycles “line up” and the paths all land in B at the same time. This can be exponentially bad as well. For example, suppose we have cycles of lengths 2, 3, 5, 7, and 11: instead of iterating just 11 times, we have to wait through $2 \cdot 3 \cdot 5 \cdot 7 \cdot 11 = 2310$ iterations for all the cycles to line up! Even worse, computing the lengths of the cycles in the first place would seem to require essentially running the original pointful GCBP algorithm, nullifying the point of the whole exercise.

Fortunately, there is a much better way to emulate at a high level what is really going on when we carry out the elementwise GCBP process. To understand it, we must first explore a few more primitive operations on partial bijections.

5 COMPATIBILITY AND MERGING

We want to think of the iteration process as monotonically revealing more and more information about the final bijection. As soon as one path completes, we have learned the fate of its starting element, and once learned, we should never “unlearn” such a fact. This motivates thinking about partial bijections in terms of their information content, and formalizing what it means for one partial bijection to be “more informative” than another. We start by formalizing some intuitive notions about partial *functions*, and then lift them to corresponding notions on partial bijections.

We say that two partial functions $f, g : A \longrightarrow \mathit{Maybe}\ B$ are *compatible*, written $f \mathbin{||} g$, if they agree at all points where both are defined, that is, for all $a : A$ and $b : B$,

$$f\ a = \mathit{Just}\ b \text{ if and only if } g\ a = \mathit{Just}\ b.$$

If two partial functions $f, g : A \longrightarrow \mathit{Maybe}\ B$ are compatible, we can define their *merge* as

$$(f \sqcup g)\ x = f\ x \diamond g\ x,$$

where the $(\diamond)$ operator from the *Alternative* type class (Figure 20) merges *Maybe* values, yielding *Nothing* if both arguments are *Nothing*, and the leftmost *Just* value otherwise. In the case of compatible partial functions, however, when both $f\ x$ and $g\ x$ are *Just*, they must be equal, so merging compatible functions does not introduce any bias—that is, when f and g are compatible, $f \sqcup g = g \sqcup f$.

These notions lift easily to the setting of partial bijections. Two partial bijections $f, g : A \rightleftharpoons B$ are compatible if their forward directions are compatible and their backward directions are compatible;

$$
\begin{array}{l}
\textbf{class } \mathit{Applicative}\ f \Rightarrow \mathit{Alternative}\ f \textbf{ where} \\
\quad \mathit{empty} :: f\ a \\
\quad (\diamond) \quad :: f\ a \to f\ a \to f\ a \\
\textbf{class } \mathit{Category}\ \mathit{arr} \Rightarrow \mathit{Mergeable}\ \mathit{arr} \textbf{ where} \\
\quad \varnothing \quad :: \mathit{arr}\ a\ b \\
\quad (\sqcup) :: \mathit{arr}\ a\ b \to \mathit{arr}\ a\ b \to \mathit{arr}\ a\ b \\
\textbf{instance } (\mathit{Monad}\ m, \mathit{Alternative}\ m) \Rightarrow \mathit{Mergeable}\ (\cdot \to_m \cdot) \textbf{ where} \\
\quad \varnothing = K\ (\mathit{const}\ \mathit{empty}) \\
\quad K\ f \sqcup K\ g = K\ \$\ \lambda a \to f\ a \diamond g\ a \\
\textbf{instance } \mathit{Mergeable}\ (\rightleftharpoons) \textbf{ where} \\
\quad \varnothing = B\ \varnothing\ \varnothing \\
\quad (B\ f\ g) \sqcup {\sim}(B\ f'\ g') = B\ (f \sqcup f')\ (g \sqcup g')
\end{array}
$$

Fig. 20. The *Mergeable* type class

the merge of two compatible partial bijections $f \sqcup g$ is computed by merging their forward and backward directions separately.

We define a type class *Mergeable*, shown in Figure 20, for categories with a monoidal structure given by a binary operator $(\sqcup)$ and a distinguished identity morphism $\varnothing$. Kleisli arrows $a \to_m b$ are *Mergeable* as long as the monad m has a suitable monoidal structure (via an *Alternative* instance), and this lifts to a corresponding instance of *Mergeable* for partial bijections. Notice that we use an *irrefutable* (*i.e.* lazy) *pattern match* in the definition of $(\sqcup)$ for partial bijections, which means $h \sqcup h'$ can output something of the form $B\ (\ldots)\ (\ldots)$ *before* demanding evaluation of h'. Evaluation of h' will only be demanded if evaluation of h on a particular input is undefined. This ensures that a chain of merged partial bijections can be evaluated lazily on each input, stopping as soon as the first partial bijection defined on the given input is found. This will be especially important in the next section, in which we will fold an *infinite* list via $(\sqcup)$.

6 GCBP VIA MERGE

As we have seen, the GCBP construction consists in starting with $h : A + B \leftrightarrow A' + B'$, and then iteratively extending it by the composite $(\varnothing + \overline{g})\ ;\ h$. Let us give a name to this iterated operation:

$\mathit{ext}_{g,h}\ k = k\ ; (\varnothing + \overline{g})\ ; h$

The leftmost column of Figure 21 shows an example of iterating this extension operation. Consider this sequence of partial bijections

$$h, \quad \mathit{ext}_{g,h}\ h, \quad \mathit{ext}^2_{g,h}\ h, \quad \mathit{ext}^3_{g,h}\ h, \quad \ldots$$

generated by iterating $\mathit{ext}_{g,h}$. That is, the first is h, the next is $h\ ; (\varnothing + \overline{g})\ ; h$, then

$$h\ ; (\varnothing + \overline{g})\ ; h\ ; (\varnothing + \overline{g})\ ; h,$$

and so on. Now we take the left projection of each, as illustrated in the rightmost column of Figure 21. In this particular example, all the left partial projections produced are compatible with each other; and moreover, the domains on which they are defined are completely disjoint. In fact, this is no coincidence: the partial bijections $\left\langle \mathit{ext}^n_{g,h}\ h \right|$ will always be pairwise compatible.

Why is this? The path an element of A takes under iteration of $\mathit{ext}_{g,h}$ can bounce around in the bottom sets (B and B'), but stops once it reaches A', since on the next iteration it will be sequenced

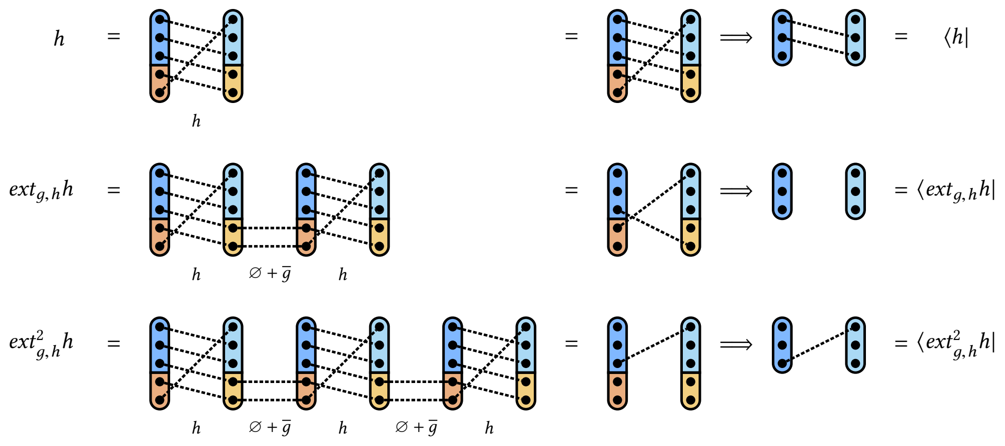

h =
h
= ⟹ = $\langle h|$
$ext_{g,h}h$ =
h $\varnothing + \overline{g}$ h
= ⟹ = $\langle ext_{g,h}h|$
$ext^2_{g,h}h$ =
h $\varnothing + \overline{g}$ h $\varnothing + \overline{g}$ h
= ⟹ = $\langle ext^2_{g,h}h|$

Fig. 21. Iterating $ext_{g,h}$

with $\varnothing$. Suppose it takes some $a \in A$ exactly n iterations to reach some $a' \in A'$. If we iterate fewer than n times, a will be mapped to some element of B', and hence the left projection will be undefined at a. If we iterate exactly n times, a will be mapped to $a' \in A'$, and hence it will map to a' in the left projection as well. If we iterate more than n times, the resulting partial bijection will be undefined at a, because after reaching a' it will be composed with $\varnothing$. So for any given $a \in A$, there is exactly one value of n such that $\left\langle ext^n_{g,h}\ h \right|$ is defined at a. Also, there can never be two different elements of A which map to the same A': two paths can never "converge" in this way since we are composing partial *bijections*, which in particular are injective where they are defined.

Hence, we are justified in considering the infinite merge

$$\left\langle h \right| \sqcup \left\langle ext_{g,h}\ h \right| \sqcup \left\langle ext^2_{g,h}\ h \right| \sqcup \left\langle ext^3_{g,h}\ h \right| \sqcup \ldots$$

For every element of A, there is some finite n for which $\left\langle ext^n_{g,h}\ h \right|$ is defined on it, and hence this infinite merge actually defines a *total* bijection, so we are justified in using *unsafeTotal* to convert it. Again, because of the irrefutable pattern match in the definition of $(\sqcup)$, this infinite expression only evaluates as far as necessary for any given input. Intuitively, this is doing exactly the same thing that the original pointwise implementation of GCBP was doing, but without having to explicitly talk about individual points $a \in A$.

With this insight, we can finally implement GCBP as follows:

$gcbp :: (a + b \leftrightarrow a' + b') \to (b \leftrightarrow b') \to (a \leftrightarrow a')$
$gcbp\ h\ g = unsafeTotal \circ foldr1\ (\sqcup) \circ map\ \left\langle \cdot \right| \circ iterate\ (ext_{g',h'})\ \$\ h'$
  where
    $g' = partial\ g$
    $h' = partial\ h$

Figure 22 demonstrates this implementation of *gcbp* using our running example.

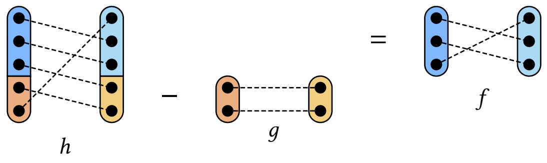
The use of *unsafeTotal* reflects the fact that the totality of the bijection constructed by *gcbp* still does not fall out "by construction"; the external reasoning outlined above is required to show that the call to *unsafeTotal* is in fact safe. On the other hand, what does fall out by construction is the fact that the resulting bijection is in fact a valid (partial) bijection, since it is constructed by composing and merging more primitive bijections. This is in contrast to the original pointwise definition of GCBP, which requires separate external reasoning to show both properties.

$$
\begin{array}{l}
\mathit{unsafeBuildBijection} :: (\mathit{Eq}\ a, \mathit{Eq}\ b) \Rightarrow [(a,b)] \rightarrow (a \leftrightarrow b) \\
\mathit{unsafeBuildBijection}\ \mathit{pairs} = \mathit{unsafeTotal}\ (f :\rightleftharpoons: g) \\
\quad \mathbf{where} \\
\qquad f = \mathit{flip}\ \mathit{lookup}\ \mathit{pairs} \\
\qquad g = \mathit{flip}\ \mathit{lookup}\ (\mathit{map}\ \mathit{swap}\ \mathit{pairs})
\end{array}
$$

$$\mathbf{data}\ \mathit{Three} = \mathit{One} \mid \mathit{Two} \mid \mathit{Three}\ \mathbf{deriving}\ (\mathit{Eq}, \mathit{Show}, \mathit{Ord}, \mathit{Enum})$$

$$
\begin{array}{l}
h :: \mathit{Three} + \mathit{Bool} \leftrightarrow \mathit{Three} + \mathit{Bool} \\
h = \mathit{unsafeBuildBijection} \\
\quad [(\mathit{Left\ One}, \quad \mathit{Left\ Two}\quad) \\
\quad ,(\mathit{Left\ Two}, \quad \mathit{Left\ Three}\ ) \\
\quad ,(\mathit{Left\ Three}, \ \mathit{Right\ False}) \\
\quad ,(\mathit{Right\ False}, \mathit{Right\ True}) \\
\quad ,(\mathit{Right\ True}, \ \mathit{Left\ One}\quad)]
\end{array}
$$

$$
\begin{array}{l}
g :: \mathit{Bool} \leftrightarrow \mathit{Bool} \\
g = \mathit{id} :\leftrightarrow: \mathit{id}
\end{array}
$$



```
> map (applyTotal (gcbp h g)) [One, Two, Three]
[Two,Three,One]
```

Fig. 22. Testing *gcbp*

7 EFFICIENCY I: MEMOIZATION

Let us return to consider the infinite merge used to compute *gcbp*:

$$\langle h | \sqcup \langle \mathit{ext}_{g,h}\ h | \sqcup \langle \mathit{ext}^2_{g,h}\ h | \sqcup \langle \mathit{ext}^3_{g,h}\ h | \sqcup \ldots$$

Operationally, to compute the output on some particular a, we perform the following series of steps:

(1) Check if $\langle h |\ a$ is defined. If yes, output it and stop.
(2) Check if $\langle \mathit{ext}_{g,h}\ h |\ a = (h\,;(\varnothing + \overline{g})\,;h)\ a$ is defined. If yes, output it and stop.
(3) Check if $\langle \mathit{ext}^2_{g,h}\ h |\ a = (h\,;(\varnothing + \overline{g})\,;h\,;(\varnothing + \overline{g})\,;h)\ a$ is defined. If yes, output it and stop.

. . . and so on. In general, in order to compute $\langle \mathit{ext}^n_{g,h}\ h |\ a$, we must evaluate an expression of size linear in n. Evaluating *applyTotal* (*gcbp h g*) *a* therefore takes time *quadratic* in the number of iterations required for a, because we first evaluate a path of length 1, then length 3, then length 5, . . . until we finally evaluate the path that takes $a \in A$ all the way to some $a' \in A'$.

As you may have already noticed, however, we are doing a lot of duplicate work. for example, to check whether $(h\,;(\varnothing + \overline{g})\,;h\,;(\varnothing + \overline{g})\,;h)\ a$ is defined, we first compute $(h\,;(\varnothing + \overline{g})\,;h)\ a$ and then apply $(\varnothing + \overline{g})\,;h$ to the result—but we already computed $(h\,;(\varnothing + \overline{g})\,;h)\ a$ on the previous iteration.

```
import           Data.MemoTrie (HasTrie)
import qualified Data.MemoTrie as MT

data KleisliM m a b where
  Id   :: KleisliM m a a
  Memo :: HasTrie a ⇒ (a → m b) → KleisliM m a b

kleisliM :: HasTrie a ⇒ (a → m b) → KleisliM m a b
kleisliM f = Memo (MT.memo f)

instance Monad m ⇒ Category (KleisliM m) where
  id = Id
  Id ∘ m = m
  m ∘ Id = m
  Memo f ∘ Memo g = kleisliM (f <=< g)

runKleisliM :: Applicative m ⇒ KleisliM m a b → (a → m b)
runKleisliM Id         = pure
runKleisliM (Memo f) = f
```

Fig. 23. Utilities for memoization

In general, the path followed on each iteration is the same as the path from the previous iteration, extended by one step.

The solution is *memoization*: along with each partial bijection, we store a map associating inputs to their corresponding outputs. corresponding outputs. With these maps in place, evaluating a partial bijection on a particular input may take any amount of time initially, but subsequent evaluations at the same input take (essentially) constant time. This means that each iteration needs to do only a constant amount of additional work, and the whole evaluation reduces to linear time instead of quadratic.

Figure 23 shows some necessary utilities for memoization. We make use of Conal Elliott's `MemoTrie` package [Elliott 2008, 2018], which provides facilities for automatically memoizing pure functions, making clever use of GHC's lazy evaluation model to store a lookup table which is lazily filled in on demand. *KleisliM* represents memoized Kleisli arrows, with a special constructor *Id* for representing the identity arrow (which does not need to be memoized). The constructor *Memo* is not exported; instead, a smart constructor *kleisliM* is provided which automatically wraps the given $a \to m\ b$ function in a call to *MT.memo*, which memoizes it. A *Category* instance for *KleisliM* defines composition, which requires another call to *MT.memo* to ensure that the result of composition is also memoized. Finally *runKleisliM* re-extracts a function of type $a \to m\ b$ from a *KleisliM m a b*. Together, *kleisliM* and *runKleisliM* witness a sort of isomorphism between $a \to m\ b$ and *KleisliM m a b*. The trick is that although *runKleisliM* (*kleisliM f*) is *semantically* equivalent to f, it is not *operationally* equivalent: in particular, evaluating *runKleisliM* (*kleisliM f*) multiple times on the same input will only evaluate f once.

Now all we have to do is redefine $\cdot \leftrightsquigarrow_m \cdot$, along with our pattern synonyms for working with total and partial bijections (Figure 24). The syntax for the pattern synonyms becomes even more horrendous, but the good news is that once we have defined them our troubles are (mostly) over: all the other code gets to stay pretty much the same, except that we have to add lots of *HasTrie* constraints everywhere, including the types of the *Parallel* and *Mergeable* methods, since we can no longer freely use any types we like, but only types for which a memo table can be built.

```
data a ↭_m b = B
  { fwd :: KleisliM m a b
  , bwd :: KleisliM m b a
  }
pattern (:⇌:) :: (HasTrie a, HasTrie b, Applicative m) ⇒ (a → m b) → (b → m a) → a ↭_m b
pattern (:⇌:) f g ← B (runKleisliM → f) (runKleisliM → g)
  where
    f :⇌: g = B (kleisliM f) (kleisliM g)
pattern (:↔:) :: (HasTrie a, HasTrie b) ⇒ (a → b) → (b → a) → a ↭_Identity b
pattern (:↔:) f g ← B (runKleisliM ; (;runIdentity) → f)
                      (runKleisliM ; (;runIdentity) → g)
  where
    f :↔: g = B (kleisliM (f ; Identity))
                (kleisliM (g ; Identity))
```

Fig. 24. Redefining $\cdot \leftrightsquigarrow_m \cdot$ with memoization

To see the difference in performance between the non-memoized and memoized versions, consider the bijection $h : A + B \leftrightarrow A + B$ which sends each element to the "next" element, except for the last element of B which gets sent back to the first element of A. Indeed, the h in our running example is of this form, with $|A| = 3$ and $|B| = 2$. Subtracting the identity bijection on B is a worst case for GCBP: although most elements in A need zero iterations to reach their final destination (*i.e.* just a single application of h), the last element of A needs $|B|$ iterations before it finishes travelling through every element of B and finally reaches its destination. Figure 25 exhibits some code which constructs this scenario, with $A = B = \{0, \ldots, 4999\}$. Figure 26 shows the result of some simple timing experiments at the GHCi prompt: without memoization, the bijection f constructed by GCBP needs 30 seconds to compute f 4999, whereas with memoization, it only needs half a second.

8 EFFICIENCY II: NAIVE COMPOSITION

However, there is still a problem: part of the point of implementing GCBP in a high-level style was so that we could compute *both* directions of the bijection at once. Memoization helps with the *forward* direction, because each iteration sequences a new partial bijection *on the right*, which means it goes *after* the part we have already computed. In the backwards direction, however, sequencing a new bijection on the right means putting it *before* the part we have already computed, and this means that we cannot reuse our previous computation. When evaluating the backward direction on a particular input, we have to start over from scratch every time.

Figure 27 has a sample GHCi session showing that memoization doesn't help with the backwards direction. Not only does it take a long time, it tends to use up all available memory so that it has to be killed.

The solution is the same as the punchline to the old joke:

Patient: Doctor, it hurts when I do this.
Doctor: Well, don't do that then.

Since adding new partial bijections on the right causes memoization to be useless for the backwards direction, let's not do that. You may protest that if we add new partial bijections on the

```
pessimal :: Integer → Integer
    → (Integer + Integer ↔ Integer + Integer, Integer ↔ Integer)
pessimal m n = (add ; cyc ; $\overline{add}$, id)
  where
      -- add :: [m] + [n] <=> [m+n]
    add = fromSum :↔: toSum
    fromSum (Left k) = k
    fromSum (Right k) = m + k
    toSum k
      | k ⩾ m = Right (k − m)
      | otherwise = Left k
    cyc = mkCyc (+1) :↔: mkCyc (subtract 1)
    mkCyc f k = f k `mod` (m + n)
```

Fig. 25. Constructing a pessimal input for GCBP

```
> let (h,g) = pessimal 5000 5000
> let f = gcbp h g
> applyTotal f 4999 -- without memoization
0
(30.40 secs, 12,135,183,272 bytes)
```

```
> let (h,g) = pessimal 5000 5000
> let f = gcbp h g
> applyTotal f 4999 -- with memoization
0
(0.55 secs, 302,438,560 bytes)
```

Fig. 26. Testing GCBP without (top) and with (bottom) memoization

```
> let (h,g) = pessimal 5000 5000
> let f = gcbp h g
> applyTotal f 4999 -- with memoization
0
(0.55 secs, 301,398,688 bytes)
> applyTotal (inverse f) 0 -- memoization doesn't help!
^CInterrupted.
```

Fig. 27. Memoization doesn't help with the backwards direction

left instead, we may fix the problem for the backwards direction but will surely reintroduce the same problem for the forwards direction, and you would be exactly right. But that isn't what we're going to do.

$$
\begin{aligned}
&\mathit{gcbp'} :: (a + b \leftrightarrow a' + b') \to (b \leftrightarrow b') \to (a \leftrightarrow a')\\
&\mathit{gcbp'}\ h\ g = \mathit{unsafeTotal} \circ \mathit{foldr1}\ (\sqcup) \circ \mathit{map}\ \big\langle \cdot \big| \circ \mathit{iterate}\ (\mathit{extendPalindrome}\ g'\ h')\ \$\ h'\\
&\quad \textbf{where}\\
&\qquad g' = \mathit{partial}\ g\\
&\qquad h' = \mathit{partial}\ h
\end{aligned}
$$

Fig. 28. Optimized *gcbp* with *extendPalindrome*

```
> let (h,g) = pessimal 5000 5000
> let f = gcbp' h g
> applyTotal f 4999
0
(0.55 secs, 301,398,688 bytes)
> applyTotal (inverse f) 0
4999
(0.25 secs, 181,462,968 bytes)
```

Fig. 29. Forwards and backwards with memoization and *extendPalindrome*

Question: when is

$$(f_1 f_2 f_3 \dots f_n)^{-1} = f_1^{-1} f_2^{-1} f_3^{-1} \dots f_n^{-1}?$$

Answer: when $f_1 \dots f_n$ is a palindrome! If $f_1 \dots f_n$ is a palindrome, it is equal to its own reversal, and hence

$$(f_1 f_2 f_3 \dots f_n)^{-1} = (f_n \dots f_3 f_2 f_1)^{-1} = f_1^{-1} f_2^{-1} f_3^{-1} \dots f_n^{-1}.$$

Luckily, GCBP is in fact computing a palindrome, namely, $h\overline{g}h\overline{g}h \dots h$. So we define an operation *extendPalindrome* $g\ h$ which in general turns $(hg)^n h$ into $(hg)^{n+1}h$, by postcomposing with another copy of g and h.

$$
\begin{aligned}
&\mathit{extendPalindrome}\\
&\quad :: (b \rightleftharpoons a) \to (a \rightleftharpoons b) \to (a \rightleftharpoons b) \to (a \rightleftharpoons b)\\
&\mathit{extendPalindrome}\ (g :\rightleftharpoons: g')\ (h :\rightleftharpoons: h')\ (f :\rightleftharpoons: f')\\
&\quad = (f\ ;\ g\ ;\ h) :\rightleftharpoons: (f'\ ;\ g'\ ;\ h')
\end{aligned}
$$

Notice how this composes f', g', and h' "backwards": one would expect

$$(f :\rightleftharpoons: f')\ ;\ (g :\rightleftharpoons: g')\ ;\ (h :\rightleftharpoons: h') = (f\ ;\ g\ ;\ h) :\rightleftharpoons: (h'\ ;\ g'\ ;\ f'),$$

but in this specific case the "naive" ordering works, since we know f is a palindrome built from g and h:

$$hg[(hg)^n h] = [(hg)^n h]gh = (hg)^{n+1}h.$$

If we redefine *gcbp* using *extendPalindrome* in place of *ext*, as shown in Figure 28, both the forwards and the backwards directions will now be sequenced together in an order which allows them to take advantage of memoization. Figure 29 has one last sample GHCi session showing that the backwards direction is now just as fast as the forwards direction.

9 CONCLUSION

In the end, what have we gained? Certainly nothing of immediate practical value: the original, pointwise implementation of GCBP is probably still the fastest, and proving it correct is not too difficult. Though it is easier to see why the high-level implementation must produce a bijection, it seems the same effort is still required to see why the produced bijection is total. And as pointed out by several reviewers, proving the correctness of the *optimized* high-level implementation is probably much *harder* than proving the pointwise implementation correct in the first place!

So why bother? This work initially grew, not out of a need for a solution or a desire to optimize, but out of a desire to *understand* the complementary bijection principle. Re-imagining the principle in a high-level, "point-free" way gives us much better insight into the original problem, its solution, and related ideas. For example, there is another principle, the celebrated *Garsia-Milne involution principle* [Garsia and Milne 1981], which turns out to be equivalent to GCBP—and this becomes very easy to see when thinking in terms of our point-free framework of partial bijections. Re-imagining GCBP at a higher level also yields potential new opportunities for generalization. For example, what happens when we choose a monad other than $m = Maybe$ for our m-bijections? Is there a deeper relationship between this work and traced monoidal categories Joyal et al. [1996], and if so, what can it tell us? Finally, the high-level construction also gives us new tradeoffs to play with when writing a mechanized formal proof; although the resulting formal proof may not be any shorter, we expect it will be more modular, with more reusable pieces (and, perhaps, more pleasant to write!). Instead of proving many tedious statements about individual elements, we can focus on proving higher-level properties of partial bijections and their operations.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their many helpful comments and suggestions. We also thank Jacques Carette for comments on a draft and pointing out some related work.

This material is based upon work supported by the National Science Foundation under Grant No. 1521539 and Grant No. 1703835. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

Conal Elliott. 2008. Elegant memoization with functional memo tries. (2008). http://conal.net/blog/posts/elegant-memoization-with-functional-memo-tries

Conal Elliott. 2018. The `MemoTrie` package. (2018). http://hackage.haskell.org/package/MemoTrie

Adriano M Garsia and Stephen C Milne. 1981. Method for constructing bijections for classical partition identities. *Proceedings of the National Academy of Sciences* 78, 4 (1981), 2026–2028.

Basil Gordon. 1983. Sieve-equivalence and explicit bijections. *Journal of Combinatorial Theory, Series A* 34, 1 (1983), 90–93.

Bjarki Ágúst Guðmundsson. 2017. *Formalizing the translation method in Agda.* Master's thesis. Reykjavik University, Iceland.

André Joyal, Ross Street, and Dominic Verity. 1996. Traced monoidal categories. In *Mathematical Proceedings of the Cambridge Philosophical Society*, Vol. 119. Cambridge University Press, 447–468.

Matthew Pickering, Gergő Érdi, Simon Peyton Jones, and Richard A Eisenberg. 2016. Pattern synonyms. In *Proceedings of the 9th International Symposium on Haskell.* ACM, 80–91.