Submitted to ICFP '12

Functional Pearl: F for Functor

Ralf Hinze Jennifer Hackett Daniel W. H. James

Department of Computer Science, University of Oxford,
Wolfson Building, Parks Road, Oxford, OX1 3QD, England
{ralf.hinze,daniel.james}@cs.ox.ac.uk, jen_hackett0@yahoo.co.uk

Abstract

Teacher: Last week I held a tutorial series with my students on the concept of a functor, with the purpose of taking a fresh look from a few angles they hadn't considered before.

Lisa: We started gently, but soon built up a heady tower of abstraction, freely wielding the force of functoriality.

Harry: For a newcomer to category theory, I can't believe how much I learnt. And we were having so much fun, we even continued on into the weekend!

Teacher: Yes, to finish off our week we looked at how we might describe the *printf* function in terms of monoidal functors.

Categories and Subject Descriptors D.1.1 [*Programming Techniques*]: Applicative (Functional) Programming; D.3.2 [*Programming Languages*]: Language Classifications—applicative (functional) languages

General Terms Theory

Keywords functor, opposite category, product category, functor category, printf, monoidal category, monoidal functor

Dramatis personæ

Teacher . . . . . A permanent resident of the ivory tower.
. . . . . [He wears socks and sandals.]
Lisa . . . . . A math student with secondary subject CS.
. . . . . [She likes a good mathematical argument.]
Harry . . . . . A Haskell geek.
. . . . . [He likes to type.]
Brooks . . . . . The Haskell expert.
. . . . . [He even has λ printed on his bedsheets.]
Bertrand . . . . . An outsider.
. . . . . [He prefers Eiffel (OO) over Haskell (FP).]
Lambert . . . . . A retired math teacher.
. . . . . [He likes to calculate.]

Dramatis locus

A small room in some medieval college building.

Monday

Teacher: Class, last week we started investigating the concept of a functor. Let's refresh our memory. Can someone summarise the salient points?

Lisa: A functor is a structure-preserving mapping between two categories—

Harry: And because a category consists of objects and arrows, a functor consists of two parts, a mapping between objects and a mapping between arrows.

Lisa: The two parts have to go together: the arrow part has to respect the types and it has to preserve identity and composition.

Teacher: That's right. Some formal definitions are certainly not amiss. If $\mathscr{C}$ and $\mathscr{D}$ are categories, then we write $\mathsf{F} : \mathscr{C} \to \mathscr{D}$ to express that F is a functor between $\mathscr{C}$ and $\mathscr{D}$. The first requirement that Lisa mentioned can be captured by two typing rules.

$$\frac{A : \mathscr{C}}{\mathsf{F}\,A : \mathscr{D}} \qquad \frac{f : A \to B : \mathscr{C}}{\mathsf{F}\,f : \mathsf{F}\,A \to \mathsf{F}\,B : \mathscr{D}}$$

Remember that we use the same symbol for both parts of the functor. The F in $\mathsf{F}\,f$ is the arrow part, the F in $\mathsf{F}\,A$ and $\mathsf{F}\,B$ is the object part. The second requirement that Lisa mentioned gives rise to two laws, the functor laws.

$$\mathsf{F}\;id = id$$
$$\mathsf{F}\;(g \cdot f) = \mathsf{F}\;g \cdot \mathsf{F}\,f$$

Where do the equations live?

Lisa: In the category $\mathscr{D}$.

Harry: This all reminds me of Haskell's *Functor* class.

$$\begin{array}{l}\textbf{class}\ \mathit{Functor}\ f\ \textbf{where}\\ \quad \mathit{map} :: (a \to b) \to (f\ a \to f\ b)\end{array}$$

Surely, this is related?

Lisa: This looks similar to the typing requirement.

Brooks: Type classes are Haskell's approach to overloading. The declaration simply expresses our intent that we wish to overload the identifier *map*. A functor is really given by a datatype or a type declaration, the object part, for instance,

$$\textbf{data}\ \mathit{Maybe}\ a = \mathit{Nothing} \mid \mathit{Just}\ a$$

and a *Functor* instance declaration, the arrow part,

$$\begin{array}{l}\textbf{instance}\ \mathit{Functor}\ \mathit{Maybe}\ \textbf{where}\\ \quad \mathit{map}\ f\ (\mathit{Nothing}) = \mathit{Nothing}\\ \quad \mathit{map}\ f\ (\mathit{Just}\ a) \quad = \mathit{Just}\ (f\ a)\end{array}$$

We don't *have* to use overloading, but it's jolly convenient.

Lisa: I can see that the class ensures that the arrow part respects the types, but I can't see how it ensures that identity and composition are preserved.

Teacher: Yes, this is a missing proof obligation—the programmer needs to convince themselves that the laws hold. What are the categories here?

Brooks: Presumably just one: Haskell's ambient category, the category with Haskell types and functions as objects and arrows.

Lisa: Isn't this a bit restrictive?

Harry: One world is enough. Why would we want more than one category for programming? Anyway, looking at functors from a programming language angle, is it fair to characterise them as container types? So, $\mathsf{F}\,A$ is an F-container of A-elements ...

Brooks: Yes, and if we continue this line of thinking, the functor laws can be seen as program optimisations.

$$map\ id = id$$
$$map\ (g \cdot f) = map\ g \cdot map\ f$$

The second equation is actually known as *map fusion*; it allows us to replace a double traversal by a single one.

[*Bertrand joins in from the sidelines*]

Bertrand: *Ah*, containers. So, in object-oriented terms, a functor F can be thought of as a generic container type, and the arrow part $\mathsf{F}\,f$ is an internal iterator.

Lisa: I'm not sure I follow. What's an internal iterator?

Bertrand: Perhaps it would be easier if I showed you what I meant. I'll use Java in deference to the Eiffel-impaired.

```
interface Function⟨T⟩ {
  void invoke (T x);
}
class Container⟨T⟩ {
  void iterate (Function⟨T⟩ f) {
    // some kind of loop that applies
    // f.invoke () to each element
  }
}
```

The *Container* class is parametrised by a type, T, and offers a method to apply a function to each element of the collection. Since Java doesn't have first-class functions like Haskell does, I've created a new type, *Function*, for wrapping up a block of code.

Lisa: But since a functor F takes an arrow $f : A \to B$ to an arrow $\mathsf{F}\,f : \mathsf{F}\,A \to \mathsf{F}\,B$, the output of the iteration shouldn't be *void*: it should be another *Container* of a different type.

Bertrand: Yes, but it's fairly simple to create a *Function* that constructs a new *Container* for you. This is more general.

Teacher: This is certainly one way to look at the concept of a functor. However, as Lisa has suggested, this is only a limited view. Remember, a categorical object is not necessarily a set, or an arrow a function, or a type and a program, for that matter. What examples of categories have we discussed previously?

Lisa: Every preorder can be seen as a category: the objects are the elements of the preorder, and there is an arrow between two objects if they are related by the preorder. This is a rather extreme example though as there is at most one arrow between any two objects.

Teacher: So a functor between two preorders seen as categories—

Lisa: Is a monotone function! We discussed another extreme example. Every monoid can be seen as a category: there is only one object, and the arrows are the elements of the monoid. Since *id* is the identity element, and the composition of arrows corresponds to the product of the monoid, a functor will be a mapping that preserves the monoid structure: a monoid homomorphism.

Teacher (*smiling*)**:** Indeed. And how about a functor between a preorder and a monoid. Consider the natural numbers. The less than or equal relation $\leq$ is a preorder, and addition with zero forms a monoid. Can you construct a functor between these categories?

Lisa: The object part is obvious as it must map every natural number to the dummy object of the monoid category: $\mathsf{F}\,n = \bullet$. It seems as though the arrow part is often where the real action happens. It must take an arrow between two natural numbers to an arrow that represents a natural number. Moreover, to preserve identity and composition, reflexivity must map to zero and transitivity must map to addition. I'll pick subtraction, or *distance*.

$$\mathsf{F}\,(n \leq m) = m - n$$

Harry: Clever—and I can see how the laws hold.

$$\begin{aligned}\mathsf{F}\,(n \leq n) &= 0\\ \mathsf{F}\,(n \leq m \leq o) &= \mathsf{F}\,(m \leq o) + \mathsf{F}\,(n \leq m)\end{aligned}$$

Lisa: If functors are mappings, then can we compose them?

Teacher: We can indeed. If we have two functors $\mathsf{F} : \mathscr{C} \to \mathscr{D}$ and $\mathsf{G} : \mathscr{D} \to \mathscr{E}$, we can compose them by composing their object and arrow parts in the obvious way: $(\mathsf{G} \circ \mathsf{F})\,A = \mathsf{G}\,(\mathsf{F}\,A)$, $(\mathsf{G} \circ \mathsf{F})\,f = \mathsf{G}\,(\mathsf{F}\,f)$. And what might complement this?

Lisa: We can construct a functor $\mathsf{Id} : \mathscr{C} \to \mathscr{C}$, for any category $\mathscr{C}$, that maps objects and arrows to themselves, $\mathsf{Id}\,A = A$, $\mathsf{Id}\,f = f$. This satisfies the laws and is the identity for functor composition.

Teacher: So if functors can be composed and have an identity—

Harry: There's a category of categories?

Lisa: Doesn't this have the same problem as the set of all sets?

Teacher: Technically you are right, Lisa. However, we can avoid size problems by restricting to *small* categories, which are categories where the collections of arrows and objects are both sets. This category of small categories is called **Cat**.

Here's a homework exercise for you: consider the category **1**, which has exactly one object and one arrow, the identity for that object. Show that there is a unique functor from any category to **1**.

Tuesday

Harry: I'd like to return to the programming language examples if you don't mind. I've been playing with monads lately and I've read that every monad is a functor. I checked the requirements for the reader monad and everything works out nicely:

$$\begin{aligned}&\textbf{type}\ \mathsf{Reader}\ r\ a = r \to a\\ &\textbf{instance}\ \mathit{Functor}\ (\mathsf{Reader}\ r)\ \textbf{where}\\ &\quad \mathit{map}\,f\ r = f \cdot r\end{aligned}$$

Teacher: That's correct; go on ...

Harry: I've noticed a library function called *with*

$$\begin{aligned}&\mathit{with} \quad :: (r' \to r) \to (\mathsf{Reader}\ r\ a \to \mathsf{Reader}\ r'\ a)\\ &\mathit{with}\,f\ r = r \cdot f\end{aligned}$$

that executes a computation in a modified environment. It's almost like a *map*, but the types aren't quite right. Perhaps, we need another concept. What about this?

$$\begin{aligned}&\textbf{class}\ \mathit{Opfunctor}\,f\ \textbf{where}\\ &\quad \mathit{opmap} :: (b \to a) \to (f\,a \to f\,b)\end{aligned}$$

I've taken the declaration from paper [9] on the reading list.

Teacher: Now I see what you mean. That's an excellent example, so let's investigate. What happens if you pass *with* the identity?

Harry: Basically nothing, we have $\mathit{with}\ id = id$.

Teacher: If you first modify the environment using f and then using g, how do the effects accumulate?

Lisa: I've done a quick calculation. It's similar to the second functor law, except that the effects are flipped: $with\ g \cdot with\ f = with\ (f \cdot g)$. In a sense, everything is the wrong way round.

Teacher: The wrong way, or the opposite way ... We can turn the data into a functor—we don't need a second concept—if we define a category where everything is flipped. Specifically, the opposite category $\mathscr{C}^{\mathsf{op}}$ has the same objects as $\mathscr{C}$; the arrows of $\mathscr{C}^{\mathsf{op}}$ are in one-to-one correspondence to the arrows in $\mathscr{C}$, that is, $f^{\mathsf{op}} : A \to B : \mathscr{C}^{\mathsf{op}}$ if and only if $f : B \to A : \mathscr{C}$.

$$\frac{A : \mathscr{C}}{A : \mathscr{C}^{\mathsf{op}}} \qquad \frac{f : B \to A : \mathscr{C}}{f^{\mathsf{op}} : A \to B : \mathscr{C}^{\mathsf{op}}}$$

Identity and composition are defined flipwise.

$$id^{\mathsf{op}} = id$$
$$f^{\mathsf{op}} \cdot g^{\mathsf{op}} = (g \cdot f)^{\mathsf{op}}$$

Ask yourself later what these rules and definitions also look like.

Harry: I don't understand. Doesn't that require that every arrow in $\mathscr{C}$ has an inverse?

Teacher: No, this is just a different view on the same data. Think of looking into a mirror.

Brooks: Or, think of a **newtype** declaration, where $(-)^{\mathsf{op}}$ is the **newtype** constructor.

Harry: *Ah*, I see. To me, it seems as though category theory is overloading the function type arrow to mean different things?

Teacher: In a sense, yes. This is why we often write $f : A \to B : \mathscr{C}$ to make explicit that f is an arrow from A to B *in* the category $\mathscr{C}$. Before I forget, a functor of type $\mathscr{C}^{\mathsf{op}} \to \mathscr{D}$ or $\mathscr{C} \to \mathscr{D}^{\mathsf{op}}$ is sometimes called a contravariant functor from $\mathscr{C}$ to $\mathscr{D}$, the usual kind being styled covariant.

Bertrand: Contravariant? *Ugh*, I've been struggling with this for years.

Brooks: If I recall correctly, variance in object-oriented languages isn't quite the same. In the case of functors we are talking about the direction of arrows, while for object-oriented languages we are really talking about the type hierarchy. Consider this example:

```
class A { }
class B extends A { }

A coA ();
B coB ();
void contraA (A x);
void contraB (B x);
```

We say *coB*'s type is a subtype of *coA*'s, as a B is also an A. In other words, *coB* can be given *coA*'s type. This respects the subtype order between A and B, so we say this is covariant.

However, *contraA*'s type is a subtype of *contraB*'s. We can use a B in place of an A, so *contraA* can be given *contraB*'s type. The order has reversed, so we say it's contravariant.

You must follow these rules when you override a method, so that the new method's type is at least as general as the old one. This reflects the principle that preconditions should only ever be made more general, while postconditions should only be made more specific.

Teacher: The concepts are more closely related than you might think. In a type hierarchy, there are implicit functions to convert from a subtype to a supertype. We call these 'inclusion arrows'. Covariance and contravariance in object-oriented languages refers to whether the functor that constructs the type is covariant or contravariant with respect to the inclusion arrows.

Bertrand: Eiffel avoids this nonsense: the subclass always deals in subclasses of what the superclass used.

Harry: But then what happens when you substitute in a subclass where a superclass was used before?

Bertrand: *Erm ...*

[*There is a brief period of silence.*]

Teacher: I think we are out of time for today. Before you all go, I'd like to set an exercise. Recall that we write the set of all arrows from A to B in a category $\mathscr{C}$ as $\mathscr{C}(A,B)$. Try to turn this into a functor from $\mathscr{C}$ to **Set**, the category of sets and functions. You should find that there are two ways of doing this: one that results in a covariant functor, and another that results in a contravariant functor.

Wednesday

Harry: I've got a question that follows on from yesterday ...

Teacher: Sure, go ahead!

Harry: So far we have concentrated on container types—this is still my preferred way to look at functors—with one parameter. But in Haskell, datatypes may have an arbitrary number of arguments. Actually, Reader is an example of a datatype with two arguments. Don't we need a generalisation of the concept of a functor to accommodate for that? For instance, binary functors

```
class Bifunctor f where
  bimap :: (a → b) → (c → d) → (f a c → f b d)
```

Again, I've taken the declaration from paper [3] on the reading list.

Teacher: OK. What properties would you expect?

Harry: Well, just two argument versions of the two functor laws: $bimap\ id\ id = id$ and $bimap\ (g_1 \cdot f_1)\ (g_2 \cdot f_2) = bimap\ g_1\ g_2 \cdot bimap\ f_1\ f_2$. [*Smiles.*] The laws were actually listed in the paper.

Lisa (*getting slightly excited*)**:** Indices, ugly. But couldn't we play the same trick as we did for contravariant functors; introducing a new category, I mean?

Teacher: Have a go!

Lisa: An object would consist of two objects and an arrow would consist of two arrows, but I'm not sure about the details.

Teacher: That was a good attempt. The category you are thinking of is called the product category. Like the opposite category, it's a construction on categories: it takes two categories, say, $\mathscr{C}$ and $\mathscr{D}$ to another category, written $\mathscr{C} \times \mathscr{D}$. A little Gedankenexperiment might help you with the details. Assume that we've succeeded in turning $\mathscr{C} \times \mathscr{D}$ into a category. Then we might wish for two projection functors, $\mathsf{Outl} : \mathscr{C} \times \mathscr{D} \to \mathscr{C}$ and $\mathsf{Outr} : \mathscr{C} \times \mathscr{D} \to \mathscr{D}$ that retrieve the original categories.

Harry: Isn't their definition pretty obvious?

$$\mathsf{Outl}\ (A,B) = A \qquad \mathsf{Outr}\ (A,B) = B$$
$$\mathsf{Outl}\ (f,g)\ = f \qquad \mathsf{Outr}\ (f,g)\ = g$$

Lisa: Yes, but the question is, what does this imply? Let's see; Outl and Outr have to respect the types.

$$\frac{P : \mathscr{C} \times \mathscr{D}}{\mathsf{Outl}\ P : \mathscr{C}} \qquad \frac{p : P \to Q : \mathscr{C} \times \mathscr{D}}{\mathsf{Outl}\ p : \mathsf{Outl}\ P \to \mathsf{Outl}\ Q : \mathscr{C}}$$

$$\frac{P : \mathscr{C} \times \mathscr{D}}{\mathsf{Outr}\ P : \mathscr{D}} \qquad \frac{p : P \to Q : \mathscr{C} \times \mathscr{D}}{\mathsf{Outr}\ p : \mathsf{Outr}\ P \to \mathsf{Outr}\ Q : \mathscr{D}}$$

So this fixes the types of the arrows in the product category?

Teacher: Indeed. We've derived the first half of the official definition of a product category. Let $\mathscr{C}$ and $\mathscr{D}$ be categories. An object of the product category $\mathscr{C} \times \mathscr{D}$ is a pair (A,B) of objects $A : \mathscr{C}$ and $B : \mathscr{D}$; an arrow of $(A,B) \to (C,D)$ is a pair (f,g) of arrows

$f : A \to C : \mathscr{C}$ and $g : B \to D : \mathscr{D}$.

$$\frac{A : \mathscr{C} \quad B : \mathscr{D}}{(A,B) : \mathscr{C} \times \mathscr{D}} \qquad \frac{f : A \to B : \mathscr{C} \quad g : C \to D : \mathscr{D}}{(f,g) : (A,C) \to (B,D) : \mathscr{C} \times \mathscr{D}}$$

How do we define identity and composition?

Lisa: Well, so far we haven't made use of the fact that Outl and Outr have to preserve identity and composition.

$$\begin{aligned} \mathsf{Outl}\ id &= id & \mathsf{Outr}\ id &= id \\ \mathsf{Outl}\ (q \cdot p) &= \mathsf{Outl}\ q \cdot \mathsf{Outl}\ p & \mathsf{Outr}\ (q \cdot p) &= \mathsf{Outr}\ q \cdot \mathsf{Outr}\ p \end{aligned}$$

Harry (*enlightened*)**:** *Aha*, this implies that the identity in $\mathscr{C} \times \mathscr{D}$ is a pair of identities and that composition composes the two components in parallel.

Teacher: Yes, they are defined componentwise:

$$\begin{aligned} id &= (id, id) \\ (h,k) \cdot (f,g) &= (h \cdot f, k \cdot g) \end{aligned}$$

Product categories avoid the need for functors of several arguments. Functors from a product category are often called bifunctors.

Harry: So all together, may I conclude that Reader is a functor of type $\mathscr{H}^{\mathsf{op}} \times \mathscr{H} \to \mathscr{H}$, where $\mathscr{H}$ is Haskell's ambient category?

Teacher: Yes, that's right.

Lisa: Stop, stop, stop. I don't see this at all! If I understand correctly, Reader r a is a covariant functor if we fix r, and it's a contravariant functor if we fix a. Are we really able to conclude that it's also a bifunctor?

Teacher: That's a very insightful remark. In general, two functors don't make a bifunctor, if you see what I mean. Assume that we have a bifunctor $\otimes : \mathscr{C} \times \mathscr{D} \to \mathscr{E}$, which we'll write infix for clarity. If we fix the first argument, we obtain another functor and likewise for the second: $A \otimes - : \mathscr{D} \to \mathscr{E}$ and $- \otimes B : \mathscr{C} \to \mathscr{E}$.

Brooks: *Ah*, operator sections.

Teacher: *Oh*, sorry; but I hope the notation is clear. Back to the topic. The converse is not true: if $A \otimes - : \mathscr{D} \to \mathscr{E}$ is a functor for each $A : \mathscr{C}$, and $- \otimes B : \mathscr{C} \to \mathscr{E}$ is a functor for each $B : \mathscr{C}$, then $\otimes$ isn't necessarily a bifunctor. Why?

Lisa: So the object part of $\otimes$ is given, but not the arrow part? The only option to define the latter is to compose the unary functors $A \otimes -$ and $- \otimes B$. Let $f : A \to A' : \mathscr{C}$ and $g : B \to B' : \mathscr{D}$, then

$$f \otimes g = (A' \otimes g) \cdot (f \otimes B) \tag{1}$$

Identity is preserved, but I'm not so sure about composition ...

[*Lambert senses an opportunity to contribute.*]

Lambert: Let's calculate, with $f' : A' \to A''$ and $g' : B' \to B''$.

$$\begin{aligned} & (f' \cdot f) \otimes (g' \cdot g) \\ = \quad & \{ \text{ definition of } \otimes \ (1) \ \} \\ & (A'' \otimes (g' \cdot g)) \cdot ((f' \cdot f) \otimes B) \\ = \quad & \{ \ B \otimes - \text{ functor and } - \otimes C \text{ functor } \} \\ & (A'' \otimes g') \cdot (A'' \otimes g) \cdot (f' \otimes B) \cdot (f \otimes B) \end{aligned}$$

Starting at the other end, we obtain

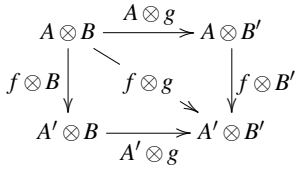
$$\begin{aligned} & (f' \otimes g') \cdot (f \otimes g) \\ = \quad & \{ \text{ definition of } \otimes \ (1), \text{ twice } \} \\ & (A'' \otimes g') \cdot (f' \otimes B') \cdot (A' \otimes g) \cdot (f \otimes B) \end{aligned}$$

To be able to connect the loose end we have to assume

$$(A'' \otimes g) \cdot (f' \otimes B) = (f' \otimes B') \cdot (A' \otimes g) \tag{2}$$

Teacher: Excellent. To define the bifunctor we can apply the unary functors in any order. The coherence condition (2) demands that the order does not matter, so we can take either side as the definition.

Harry: Here, I find a little diagram always helps me:



Lisa: OK, I'm much happier now. By the way, Reader passes the coherence check, so yes, it *is* a bifunctor.

Teacher: Here is a homework exercise for you: conduct a Gedankenexperiment to discover the structure of a coproduct category, assuming functors $\mathsf{Inl} : \mathscr{C} \to \mathscr{C} + \mathscr{D}$ and $\mathsf{Inr} : \mathscr{D} \to \mathscr{C} + \mathscr{D}$.

Thursday

Harry: In Haskell, datatypes may have parametric types as arguments, if that makes sense. No? OK, I'd better give an example.

$$\mathbf{data}\ \mathsf{Rose}\ f\ a = \mathit{Node}\ a\ (f\ (\mathsf{Rose}\ f\ a))$$

This defines a general tree type where we've abstracted away from the subtree structure: f is a container type, so a tree node consists of an a-label and an f-structure of subtrees.

Teacher: Are you suggesting that f is a functor?

Harry: Possibly; at least, I need to make this assumption when defining the functor instance for Rose.

$$\begin{aligned} &\mathbf{instance}\ (\mathit{Functor}\ f) \Rightarrow \mathit{Functor}\ (\mathsf{Rose}\ f)\ \mathbf{where} \\ &\quad \mathit{map}\ f\ (\mathit{Node}\ a\ ts) = \mathit{Node}\ (f\ a)\ (\mathit{map}\ (\mathit{map}\ f)\ ts) \end{aligned}$$

I'm tempted to conclude that Rose is a higher-order functor, a functor of type $(\mathscr{H} \to \mathscr{H}) \to (\mathscr{H} \to \mathscr{H})$.

Lisa: This doesn't make any sense as $\mathscr{H} \to \mathscr{H}$ is not a category!

Harry (*disappointed*)**:** Too bad.

Brooks: In Haskell, the type of a type is described using the kind system. The type Rose has kind $(* \to *) \to (* \to *)$. Maybe Harry is on the right track after all.

Teacher: Well, we can unloose the Gordian Knot if we manage to turn $\mathscr{H} \to \mathscr{H}$ into a category. So, let's do that! Generalising slightly, the task is to impose categorical structure on $\mathscr{C} \to \mathscr{D}$. Actually, to avoid confusion, we should use a different notation for this category: $\mathscr{D}^{\mathscr{C}}$ is standard.

Lisa: The objects of $\mathscr{D}^{\mathscr{C}}$ are certainly functors—

Harry: And functors compose; we are done!

Lisa: No, no. We need to find out what the arrows are—they need to compose, not the objects.

Harry (*flushes a bit*)**:** Yes, you are right.

Teacher: Let's conduct another Gedankenexperiment. Assume that we've succeeded in turning $\mathscr{D}^{\mathscr{C}}$ into a category. The next logical step would be to turn functor application into a functor.

Harry: Functor application's a functor? My brain just exploded ...

Lisa (*pondering*)**:** Let's first make the categories explicit, after all, a functor is a mapping between two categories. Functor application has type $\mathscr{D}^{\mathscr{C}} \times \mathscr{C} \to \mathscr{D}$ ... so functor application is a bifunctor.

Harry: I guess you are telling me that we should apply what we learnt yesterday about bifunctors?

Lisa: Harry, that's brilliant; we simply start at the other end by turning $\mathsf{F} - : \mathscr{C} \to \mathscr{D}$ and $- A : \mathscr{D}^{\mathscr{C}} \to \mathscr{D}$ into unary functors—

Harry: And then check the coherence property for bifunctors!? Your notation is strange: is it derived from $\mathsf{F}\ A$, applying F to A?

Lisa: That's right. Now, $\mathsf{F} - : \mathscr{C} \to \mathscr{D}$ is just F, so this is a functor by definition. The second one, $- A : \mathscr{D}^{\mathscr{C}} \to \mathscr{D}$, looks more

interesting. To be able to turn it into a functor, $\mathscr{D}^{\mathscr{C}}$ has to be a category. Perhaps this tells us something about the arrows of $\mathscr{D}^{\mathscr{C}}$? Remember, functors have to respect the types:

$$\frac{\mathsf{F} : \mathscr{D}^{\mathscr{C}}}{\mathsf{F}\,A : \mathscr{D}} \qquad \frac{\alpha : \mathsf{F} \to \mathsf{G} : \mathscr{D}^{\mathscr{C}}}{\alpha\,A : \mathsf{F}\,A \to \mathsf{G}\,A : \mathscr{D}}$$

This suggests that an arrow $\alpha : \mathsf{F} \to \mathsf{G}$ in $\mathscr{D}^{\mathscr{C}}$ sends an object A of $\mathscr{C}$ to an arrow $\alpha\,A : \mathsf{F}\,A \to \mathsf{G}\,A$ in $\mathscr{D}$.

Harry: That's not very informative if you ask me.

Lisa: Because we've not yet applied the force of functoriality. The functor $-\,A$ also has to preserve identity and composition.

$$id\,A = id$$
$$(\beta \cdot \alpha)\,A = \beta\,A \cdot \alpha\,A$$

We can take these equations as the definitions of identity and composition in $\mathscr{D}^{\mathscr{C}}$!

Harry: That's all? An arrow in $\mathscr{D}^{\mathscr{C}}$ sends an object to an arrow, and identity and composition are defined 'pointwise'?

Lisa: Not quite. As you've pointed out, we also have to check the coherence condition. Let's see. If $\alpha : \mathsf{F} \to \mathsf{G}$ and $f : A \to B$, then

$$\mathsf{G}\,f \cdot \alpha\,A = \alpha\,B \cdot \mathsf{F}\,f$$

Harry: OK. Let me try to get this straight in my head: F and G are container types; α transforms F-containers into G-containers. The coherence condition says that we can either map f over the F-structure and then transform, or transform first and then map f over the resulting G-structure.

Teacher: I continue to be impressed! You have just discovered the concept of a *natural transformation*. That's the official term for the arrows in $\mathscr{D}^{\mathscr{C}}$. The category itself is called a functor category, because the objects are functors. Let me summarise.

Let $\mathsf{F}, \mathsf{G} : \mathscr{C} \to \mathscr{D}$ be two parallel functors. A *transformation* $\alpha : \mathsf{F} \to \mathsf{G}$ is a collection of arrows, so that for each object $A : \mathscr{C}$ there is an arrow $\alpha\,A : \mathsf{F}\,A \to \mathsf{G}\,A : \mathscr{D}$. A transformation is a mapping from objects to arrows. The transformation α is *natural*, $\alpha : \mathsf{F} \dot{\to} \mathsf{G}$, if

$$\mathsf{G}\,h \cdot \alpha\,A = \alpha\,B \cdot \mathsf{F}\,h \tag{3}$$

for all arrows $h : A \to B : \mathscr{C}$. Given α and h, there are essentially two ways of turning $\mathsf{F}\,A$ things into $\mathsf{G}\,B$ things. The coherence condition (3) demands that they are equal.

$$\begin{array}{ccc} \mathsf{F}\,A & \xrightarrow{\mathsf{F}\,h} & \mathsf{F}\,B \\ {\scriptstyle \alpha\,A}\downarrow & & \downarrow{\scriptstyle \alpha\,B} \\ \mathsf{G}\,A & \xrightarrow[\mathsf{G}\,h]{} & \mathsf{G}\,B \end{array}$$

With natural transformations $id_{\mathsf{F}} : \mathsf{F} \dot{\to} \mathsf{F}$, $\alpha : \mathsf{F} \dot{\to} \mathsf{G}$, and $\beta : \mathsf{G} \dot{\to} \mathsf{H}$, identity and composition are defined pointwise:

$$id_{\mathsf{F}}\,A = id_{\mathsf{F}\,A}$$
$$(\beta \cdot \alpha)\,A = \beta\,A \cdot \alpha\,A$$

Thus, functors of type $\mathscr{C} \to \mathscr{D}$ and natural transformations between form the functor category $\mathscr{D}^{\mathscr{C}}$.

$$\frac{\mathsf{F} : \mathscr{C} \to \mathscr{D}}{\mathsf{F} : \mathscr{D}^{\mathscr{C}}} \qquad \frac{\alpha : \mathsf{F} \dot{\to} \mathsf{G}}{\alpha : \mathsf{F} \to \mathsf{G} : \mathscr{D}^{\mathscr{C}}}$$

Harry: Let's get back to my example, rose trees. Just to be clear, the arrow part of Rose has to send natural transformations to natural transformations. But how I would capture this in Haskell?

Teacher: As a first approximation you can equate natural transformations with polymorphic functions.

Harry (*starts typing*)**:** Of course! So the arrow part of Rose is a rank-2 function, as it takes a polymorphic function as an argument.

```
class Hofunctor h where
  homap :: (∀a . f a → g a) → (∀a . h f a → h g a)
instance Hofunctor Rose where
  homap α (Node a ts) = Node a (map (homap α) (α ts))
```

This doesn't quite work, the compiler complains about a missing *Functor g* instance. Swapping *map* (*homap* α) and α,

```
instance Hofunctor Rose where
  homap α (Node a ts) = Node a (α (map (homap α) ts))
```

results in a missing *Functor f* instance.

Brooks: The type of *homap* is too strong. You will need to assume that f and g are functors.

```
class Hofunctor h where
  homap :: (Functor f, Functor g) ⇒
           (∀a . f a → g a) → (∀a . h f a → h g a)
```

[*Peeks over Harry's shoulder.*]
Incidentally, the error message suggests a fix along those lines.

Teacher: Excellent. That wasn't quite what I intended to cover today, but I think it was a very good discussion nonetheless. To wrap up, the parametric type Rose can indeed be seen as a functor, a functor of type $\mathscr{H}^{\mathscr{H}} \to \mathscr{H}^{\mathscr{H}}$. Again, the important point is that a new concept isn't needed: Harry's hofunctor, a contraction of the unwieldy term higher-order functor, I suppose, is just a functor between functor categories. You can find a similar type class in paper [6] on the reading list.

Before you run away, here is a homework exercise for you: use the characterisation of bifunctors to turn functor composition into a functor of type $\mathscr{E}^{\mathscr{D}} \times \mathscr{D}^{\mathscr{C}} \to \mathscr{E}^{\mathscr{C}}$.

Friday

Harry: I guess it's me kicking off the discussion again; I'm not sure I've fully digested the idea of functor application as a functor.

Teacher: Well, we haven't actually spelled out its definition. Perhaps this is a good opportunity to close the gap. Maybe it's also helpful to introduce an explicit operator for the 'functor application' functor, say $\star$. Happy with that?

Harry: Not really. What's the difference between $\mathsf{F}\,A$ and $\mathsf{F} \star A$?

Lisa: The former is standard notation—well, at least in FP—for applying a mapping to an argument. A functor consists of two mappings; in $\mathsf{F}\,A$ the object part of F is applied to an argument, the object A. By contrast, in $\mathsf{F} \star A$ the functor F is an *argument*, the mapping is $\star$, a functor whose object part is applied to two objects: F, which is an object in a functor category, and the object A.

Harry: Thanks, Lisa; let me try to work out the typing rules.

$$\frac{\mathsf{F} : \mathscr{D}^{\mathscr{C}} \quad A : \mathscr{C}}{\mathsf{F} \star A : \mathscr{D}} \qquad \frac{\alpha : \mathsf{F} \to \mathsf{G} : \mathscr{D}^{\mathscr{C}} \quad f : A \to B : \mathscr{C}}{\alpha \star f : \mathsf{F} \star A \to \mathsf{G} \star B : \mathscr{D}}$$

Teacher: Excellent. And how are you going to define $\star$?

Lisa: The object part is clear: we apply the functor to the object. For the arrow part, we build on the characterisation of bifunctors, using either side of the coherence condition (3).

$$\mathsf{F} \star A = \mathsf{F}\,A$$
$$\alpha \star h = \mathsf{G}\,h \cdot \alpha\,A = \alpha\,B \cdot \mathsf{F}\,h \tag{4}$$

Harry: OK, *now* I see the difference between $\mathsf{F} \star A$ and $\mathsf{F}\,A$.

Brooks: We've covered functor application, but is there also such a thing as functor abstraction?

Harry: Abstraction as the inverse of application?

Teacher: The notion of an inverse isn't quite appropriate: you can't recreate function and argument from the result of a function application. But application has what is sometimes called a quasi-inverse. Specialised to functors: for each functor $\mathsf{F} : \mathscr{C} \times \mathscr{D} \to \mathscr{E}$ there exists a functor $\Lambda\,\mathsf{F} : \mathscr{C} \to \mathscr{E}^{\mathscr{D}}$ such that

$$\left.\begin{array}{l}\mathsf{F}\,(A,B) = \mathsf{G}\,A \star B \\ \mathsf{F}\,(f,g)\ = \mathsf{G}\,f \star g\end{array}\right\} \iff \Lambda\,\mathsf{F} = \mathsf{G} \tag{5}$$

for all $\mathsf{G} : \mathscr{C} \to \mathscr{E}^{\mathscr{D}}$. So Λ turns a bifunctor into a hofunctor. Actually, this should appeal to you, Brooks, it's called currying.

Brooks: You mean currying as in turning a function of two arguments into a function of the first argument whose values are functions of the second argument?

Teacher: Exactly, but on the level of functors.

Harry: But then the definition of $\Lambda\,\mathsf{F}$ is straightforward, isn't it: $\Lambda\,\mathsf{F}\,A\,B = \mathsf{F}\,(A,B)$ and $\Lambda\,\mathsf{F}\,f\,g = \mathsf{F}\,(f,g)$.

Teacher: Not quite, $\Lambda\,\mathsf{F}$ has to be functorial, but also $\Lambda\,\mathsf{F}\,A$. Let's spell out the categories involved.

$$\frac{A : \mathscr{C}}{\Lambda\,\mathsf{F}\,A : \mathscr{E}^{\mathscr{D}}} \qquad \frac{f : A \to A' : \mathscr{C}}{\Lambda\,\mathsf{F}\,f : \Lambda\,\mathsf{F}\,A \to \Lambda\,\mathsf{F}\,A' : \mathscr{E}^{\mathscr{D}}}$$

Lisa: In the arrow part, $\Lambda\,\mathsf{F}\,f$ is a natural transformation that sends an object of $\mathscr{D}$ to an arrow in $\mathscr{E}$. What about $\Lambda\,\mathsf{F}\,f\,B = \mathsf{F}\,(f, id_B)$?

Teacher: Looks good to me. And the object part?

Lisa: Because $\Lambda\,\mathsf{F}\,A$, in turn, has to be a functor, it consists of an object and an arrow part.

$$\frac{B : \mathscr{D}}{\Lambda\,\mathsf{F}\,A\,B : \mathscr{E}} \qquad \frac{g : B \to B' : \mathscr{D}}{\Lambda\,\mathsf{F}\,A\,g : \Lambda\,\mathsf{F}\,A\,B \to \Lambda\,\mathsf{F}\,A\,B' : \mathscr{E}}$$

For the object part we can actually use Harry's original definition: $\Lambda\,\mathsf{F}\,A\,B = \mathsf{F}\,(A,B)$. And the arrow part is $\Lambda\,\mathsf{F}\,A\,g = \mathsf{F}\,(id_A, g)$.

Harry: *Yikes*, this is complicated.

Teacher: Let's bring the definitions all together.

$$\begin{aligned}\Lambda\,\mathsf{F}\,A\,B &= \mathsf{F}\,(A,B)\\ \Lambda\,\mathsf{F}\,A\,g &= \mathsf{F}\,(id_A, g)\\ \Lambda\,\mathsf{F}\,f\ B &= \mathsf{F}\,(f, id_B)\end{aligned}$$

To see what's what, it's helpful to name the intermediate functors.

$$\Lambda\,\mathsf{F} = \mathsf{G}\ \textbf{where}\ \begin{cases}\mathsf{G}\,A\ = \mathsf{H}\ \textbf{where}\ \begin{cases}\mathsf{H}\,B = \mathsf{F}\,(A,B)\\ \mathsf{H}\,g\ = \mathsf{F}\,(id_A, g)\end{cases}\\ \mathsf{G}\,f\,B = \mathsf{F}\,(f, id_B)\end{cases} \tag{6}$$

So Λ sends F to G, which in turn sends A to H. A number of proof obligations arise: we have to show that $\mathsf{G}\,f$ is natural and that G and H preserve identity and composition. The proofs are all straightforward but I encourage you to go through the details. And you should check that (5) holds.

[*Someone wanders in, having misread the room number.*]

Olivier: *Uhh*, is this the tutorial on "Lambda Calculus and Types"? Wasn't that last term?

Saturday

Teacher: We've covered quite a lot of material over the week, so maybe we could do with a brief recap. On Monday we introduced **Cat**, the category of all small categories and functors between them. In a sense, we spent the following days exploring its structure. On Wednesday, we introduced product categories, and then functor categories on Thursday. The upshot is that **Cat** is a so-called Cartesian-closed category, CCC for short. (As an aside, Wednesday's homework exercise asked you to define coproduct categories, so **Cat** is actually bicartesian closed.) We'll take a closer look at Cartesian closure in the weeks ahead, but for now, I'll just say that Cartesian-closed categories are intimately related to simply-typed lambda calculi.

Harry (*grins*)**:** So Olivier wasn't confused yesterday, just lost.

Teacher (*serious*)**:** No, not at all. We can interpret a simply-typed lambda term in a Cartesian-closed category. In general, the interpretation of a λ-term is an arrow from the interpretation of its free variables to the interpretation of its type.

Lisa: If we pick **Cat** as the target category, then we can see a λ-term as a functor?

Teacher: Exactly. The slogan is: a λ-term is a functorial in its free variables. Let's have a look at some examples. As a warm-up, what functor is this?

$$x : C \vdash x : C$$

Harry: *Aah*, I remember this from "Lambda Calculus and Types". The sentence $x_1 : T_1, \ldots, x_n : T_n \vdash e : T$ says that if x_1 has type T_1 and so forth, then e has type T. But now, e denotes an object and T a category, is that right?

Lisa: Seems like it. So, the λ-term $x : C \vdash x : C$ presumably denotes the identity functor $\mathsf{Id} : \mathscr{C} \to \mathscr{C}$, assuming that $\mathscr{C}$ is the interpretation of C.

Teacher: OK, how about ...

$$g : D \to E, f : C \to D \vdash \lambda\, x\,.\, g\,(f\,x) : C \to E$$

Harry: So, $C \to E$ has to be interpreted as a category?

Lisa (*excited*)**:** *Ooo*, this is about what we did on Thursday, when we introduced functor categories to be able to give a meaning to higher-order functors. In which case, your λ-term is a functor of type $\mathscr{E}^{\mathscr{D}} \times \mathscr{D}^{\mathscr{C}} \to \mathscr{E}^{\mathscr{C}}$. Functor composition?

Teacher: Bingo!

Harry: Wait, we had 'functor application' as a functor and now we have 'functor composition' as a functor?

Teacher: That's right. But before we go any further, I'd like to point out why the simply-typed λ-calculus is so convenient for describing functors. In a sense, we only describe the action on objects and get the action on arrows for free.

Harry: So what *is* the action of functor composition on arrows?

Lisa: First of all, functor composition acts on two natural transformations—those are the arrows in a functor category. But the definition ... I reckon we have to combine functor abstraction and functor application. Let's break it down and consider the body of the λ-abstraction first. The typing rule for λ-abstraction

$$\frac{\Gamma, x : t \vdash e : u}{\Gamma \vdash \lambda\, x\,.\, e : t \to u}$$

suggests that we have to consider

$$g : D \to E, f : C \to D, x : C \vdash g\,(f\,x) : E$$

The interpretation of $g\,(f\,x)$ is then a functor B with $\mathsf{B}\,(\beta, \alpha, h) = \beta \star (\alpha \star h)$, functor application nested ... but I'm not quite sure how to interpret the λ-abstraction.

Harry: We've got to move a type from the left side to the right ... can't we use currying?

Lisa: Of course! So the interpretation of $\lambda\, x\,.\, g\,(f\,x)$ is just $\Lambda\,\mathsf{B}$. Using the definition of currying (6), we have $\Lambda\,\mathsf{B} = \mathsf{C}$ where $\mathsf{C}\,(\beta, \alpha)\,A = \beta \star (\alpha \star id_A)$.

Teacher: Well done, you made it. If we use infix notation for functor composition, then this translates to

$$(\beta \circ \alpha)\,A = \beta \star (\alpha \star id_A)$$

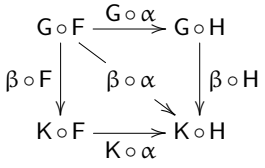
where $\alpha : \mathsf{F} \dot{\to} \mathsf{H}$ and $\beta : \mathsf{G} \dot{\to} \mathsf{K}$. Generalising slightly, we have

$$(\beta \circ \alpha) \star h = \beta \star (\alpha \star h)$$

which nicely relates functor composition and functor application.

Lisa: *Hmm*, functor composition as a functor was Thursday's homework, and I ended up with a different definition. Using the characterisation of bifunctors I looked at $\mathsf{G} \circ -$ and $- \circ \mathsf{F}$. Starting out, it wasn't clear that these were really functors. For the action on arrows I defined $(\mathsf{G} \circ \alpha)\ A = \mathsf{G}\ (\alpha\ A)$ and $(\beta \circ \mathsf{F})\ A = \beta\ (\mathsf{F}\ A)$ and everything worked out nicely. The coherence condition (2) followed from naturality, so I concluded

$$\beta \circ \alpha = (\mathsf{K} \circ \alpha) \cdot (\beta \circ \mathsf{F}) = (\beta \circ \mathsf{H}) \cdot (\mathsf{G} \circ \alpha)$$



Lambert: Calculemus igitur.

$$\begin{aligned}
& \beta \star (\alpha \star id_A) \\
= \quad & \{ \text{ definition of functor application (4) } \} \\
& \beta \star (\alpha\ A \cdot \mathsf{F}\ id_A) \\
= \quad & \{ \ \mathsf{F} \text{ functor } \} \\
& \beta \star \alpha\ A \\
= \quad & \{ \text{ definition of functor application (4) } \} \\
& \beta\ (\mathsf{H}\ A) \cdot \mathsf{G}\ (\alpha\ A) \\
= \quad & \{ \text{ definition of } \mathsf{G} \circ - \text{ and } - \circ \mathsf{H}, \text{ see above } \} \\
& (\beta \circ \mathsf{H})\ A \cdot (\mathsf{G} \circ \alpha)\ A \\
= \quad & \{ \text{ composition of natural transformations } \} \\
& ((\beta \circ \mathsf{H}) \cdot (\mathsf{G} \circ \alpha))\ A
\end{aligned}$$

Consequently, $\beta \star (\alpha \star id_A) = (\beta \circ \mathsf{H}) \cdot (\mathsf{G} \circ \alpha)$ as desired.

Harry: Hold up, I thought we already had a way to compose natural transformations, $\beta \cdot \alpha$?

Teacher: Let's have a look at the typing rules:

$$\frac{\mathsf{G} : \mathscr{E}^{\mathscr{D}} \quad \mathsf{F} : \mathscr{D}^{\mathscr{C}}}{\mathsf{G} \circ \mathsf{F} : \mathscr{E}^{\mathscr{C}}} \qquad \frac{\beta : \mathsf{G} \to \mathsf{K} : \mathscr{E}^{\mathscr{D}} \quad \alpha : \mathsf{F} \to \mathsf{H} : \mathscr{D}^{\mathscr{C}}}{\beta \circ \alpha : \mathsf{G} \circ \mathsf{F} \to \mathsf{K} \circ \mathsf{H} : \mathscr{E}^{\mathscr{C}}}$$

As you can see, the natural transformations α and β are in different functor categories. Before, we were composing natural transformations in the same functor category. The natural transformation $\beta \cdot \alpha$ is sometimes called the *vertical composition* of α and β, while $\beta \circ \alpha$ is the *horizontal composition*. We can easily see the relationship between them if we write the following equation in 2D:

$$\begin{pmatrix} (\beta_1 \circ \alpha_1) \\ \cdot \\ (\beta_2 \circ \alpha_2) \end{pmatrix} = \begin{pmatrix} \beta_1 \\ \cdot \\ \beta_2 \end{pmatrix} \circ \begin{pmatrix} \alpha_1 \\ \cdot \\ \alpha_2 \end{pmatrix}$$

This is known as the *abide law* (above and beside), and follows from the functoriality of $\circ$. (A little Bird told me about this law.)

Brooks: So, if any λ-term can be considered a functor, what about Haskell types? In Haskell, we can apply any type constructor to any type. We can only abstract types by creating a definition, but this isn't much of a restriction. We could implement functor composition in Haskell like so:

$$\textbf{type}\ (f \circ g)\ a = f\ (g\ a)$$

So, can we always turn a Haskell type constructor into a functor?

Teacher: Almost. First of all, we can add constants to the simply-typed lambda calculus to model the datatype features of the programming language at hand. We only have to be able to interpret these constants as functors. However, with regards to Haskell there is a caveat. Consider the following definition:

$$\textbf{type}\ \mathsf{R}\ a = \mathsf{Reader}\ a\ a$$

Recall the Reader functor from Tuesday and Wednesday, which we know is a bifunctor of type $\mathscr{C}^{\mathsf{op}} \times \mathscr{C} \to \mathscr{C}$. It would seem that, since we are simply applying this to some arguments, we should have a functor. We could try to define the action on arrows in terms of Reader like so:

$$\mathsf{R} f = \mathsf{Reader}\ f\ f$$

However, since Reader takes one arrow from $\mathscr{C}$ and one from $\mathscr{C}^{\mathsf{op}}$, this is only defined for arrows that are in both $\mathscr{C}$ and $\mathscr{C}^{\mathsf{op}}$: the invertible arrows.

The problem is that Haskell's kind system doesn't distinguish between the categories $\mathscr{H}$ and $\mathscr{H}^{\mathsf{op}}$. However, any Haskell type that is formed from purely covariant operations is functorial.

Harry: What about product types and sum types? Can we model those as functors?

$$\begin{aligned}
&\textbf{data}\ \mathsf{Product}\ a\ b = \mathit{Pair}\ a\ b \\
&\textbf{data}\ \mathsf{Sum}\ a\ b \quad\ = \mathit{Left}\ a \mid \mathit{Right}\ b
\end{aligned}$$

Teacher: We can model product and sum types with categorical products and coproducts, which are roughly analogous to the Cartesian product and disjoint union of sets. So for Haskell we would need to include, say, $\times$ and $+$ as constants in our calculus. Then Product and Sum correspond to

$$a : C, b : C \vdash a \times b : C$$
$$a : C, b : C \vdash a + b : C$$

We'll be talking about products and coproducts in later weeks.

Lisa: Just one quick question before you move on: are products and coproducts related to the product and coproduct categories?

Teacher: They are: the product category is the categorical product in **Cat**, and similarly for the coproduct category.

Harry: What about recursive datatypes?

$$\textbf{data}\ \mathsf{Tree}\ a = \mathit{Nil} \mid \mathit{Fork}\ a\ (\mathsf{Tree}\ a)\ (\mathsf{Tree}\ a)$$

Teacher: We can think of a recursive datatype as being the fixed point of a functor. There are actually two ways to define such a fixed point: *initial algebras* and *final coalgebras*. Again, we'll get to this in the coming weeks.

I'd like to hold an optional tutorial tomorrow where we look at an application of all this theory. I have some ideas, but please let me know if you want to look at anything in particular.

Sunday

Teacher: I'm delighted that you're all back for more today. I promised to look at an application for today and Harry suggested the *printf* function. Harry, why don't you start us off?

Harry: The challenge is to define a type-safe version of C's *printf* function, which takes a variable number of arguments.

[*Harry types out some examples on his laptop.*]

```
≫ printf nil
""
≫ printf int 4711
"4711"
≫ printf (string ◊ int) "Eau de Cologne: " 4711
"Eau de Cologne: 4711"
≫ printf (lit "Eau de Cologne: " ◊ int) 4711
"Eau de Cologne: 4711"
```

There've been a few variations on a typed *printf*, and one approach is to model the directives as values of type $\mathsf{F}\ \mathit{String}$, where the functor F specifies whether further arguments are expected or not [4].

$$\begin{aligned}&\mathit{int} \quad :: (\mathsf{Reader}\ \mathit{Int})\ \mathit{String}\\&\mathit{int}\ n = \mathit{show}\ n\\&\mathit{string} \quad :: (\mathsf{Reader}\ \mathit{String})\ \mathit{String}\\&\mathit{string}\ s = s\\&\mathit{lit} \quad :: \mathit{String} \to \mathsf{Id}\ \mathit{String}\\&\mathit{lit}\ s \ = s\end{aligned}$$

For *string*, the functor is $\mathsf{Reader}\ \mathit{String}$, which specifies that an additional argument of type *String* is required. In contrast, *lit s* has type $\mathsf{Id}\ \mathit{String}$, which means that nothing further is expected.

The directives are concatenated with $\lozenge$, which has an intriguing type and definition.

$$\begin{aligned}&\mathit{nil} :: \mathsf{Id}\ \mathit{String}\\&\mathit{nil} = \texttt{""}\\&\textbf{infixr}\ \lozenge\\&(\lozenge) \ :: (\mathit{Functor}\ g, \mathit{Functor}\ f) \Rightarrow\\&\qquad g\ \mathit{String} \to f\ \mathit{String} \to (g \circ f)\ \mathit{String}\\&x \lozenge y = \mathit{map}\ (\lambda s \to \mathit{map}\ (\lambda t \to s +\!\!+ t)\ y)\ x\end{aligned}$$

The operator composes the two functors and at the same time concatenates the strings. For this to all make sense, we need to show that $\lozenge$ is associative with *nil* as its neutral element.

Teacher: OK. First of all, let me forewarn you: this might get tricky, so don't expect to digest everything in one go. Second, I'd like to generalise the problem slightly—this will make the problem easier, not harder. The idea is to separate the monoid operations, `""` and $+\!\!+$, from the structural operations:

$$\begin{aligned}&\mathit{unit} :: \mathsf{Id}\ ()\\&\mathit{unit} = ()\\&(\circledast) \ :: (\mathit{Functor}\ g, \mathit{Functor}\ f) \Rightarrow g\ a \to f\ b \to (g \circ f)\ (a,b)\\&x \circledast y = \mathit{map}\ (\lambda a \to \mathit{map}\ (\lambda b \to (a,b))\ y)\ x\end{aligned}$$

I've turned $\lozenge$ into a polymorphic function $\circledast$, which pairs the data. As an example, if g and f are lists, then $\circledast$ is the cross product.

```
>> ['A'..'D'] ⊛ [0..2]
[[('A',0),('A',1),('A',2)],[('B',0),('B',1),('B',2)],
 [('C',0),('C',1),('C',2)],[('D',0),('D',1),('D',2)]]
```

I hope everyone can see how to define $\lozenge$ in terms of $\circledast$: if a and b are strings, then we map string concatenation over $g \circ f$. So now we'll show that $\circledast$ is associative with *unit* as its neutral element.

Harry: *Hmm*, I don't see how we're going to turn $\lozenge$ or $\circledast$ into category theory. The two occurrences of *map* are strangely nested.

Teacher: Yes, there's a subtle problem with the definition of $\circledast$: the function that's passed to the inner *map* refers to the variable a, which is bound at an outer scope. So, we can't translate this into the first-order language of category theory, unless we make some additional assumptions.

Lisa: The right-hand side of $\circledast$'s definition looks like a standard λ-term to me. We discussed yesterday that we can interpret a λ-term in a Cartesian-closed category. Is this the additional assumption?

Teacher: That's a good point. The infrastructure of a CCC gives us the machinery for manipulating environments, but that alone isn't going to cut it. Loosely speaking, the plumbing of environments also has to work across functors, yet g and f are nothing special.

Harry: How special do they need to be?

Teacher: One approach is to assume a broadcast operation $\mathit{bcl} : \mathsf{F}\ A \times B \to \mathsf{F}\ (A \times B)$, which broadcasts its second argument over its first. This has to work for all A and B, so we require *bcl* to be natural in A and B.

Harry: *bc* like the BBC?

Teacher (*chuckling*)**:** Yes, and *l* to indicate that the F-structure is on the left. (As an aside, *bcl* is also called tensorial strength.)

Harry: In Haskell I'd write:

$$\begin{aligned}&\mathit{bcl} \quad :: (\mathit{Functor}\ f) \Rightarrow (f\ a, b) \to f\ (a,b)\\&\mathit{bcl}\ (s,b) = \mathit{map}\ (\lambda\, a \to (a,b))\ s\end{aligned}$$

Teacher: That's exactly the point. The function passed to *map* refers to the variable b, which is bound at an outer scope.

Harry: *Aah*, I see.

Lisa: Sorry, I'm a bit slow; I'm still pondering about the naturality requirement. I can see that *bcl*'s source and target type are functorial in A and B—we can express both as λ-terms, which as we know are functorial in their free variables. This would imply that *bcl* is a natural transformation between bifunctors, a *binatural transformation*? Is there more to it—like coherence for bifunctors?

Teacher: Good point. Fortunately, the situation is a lot simpler for 'binatural transformations': a transformation is natural in two arguments if and only if it's natural in each argument separately. Can someone state the naturality condition for *bcl*?

Harry: What about this?

$$\mathsf{F}\ (h \times k) \cdot \mathit{bcl}_{\mathsf{F}} = \mathit{bcl}_{\mathsf{F}} \cdot (\mathsf{F}\ h \times k)$$

In the tradition of Haskell, I've omitted the type arguments of $\mathit{bcl}_{\mathsf{F}}$.

Lisa: Yes, and the functor laws imply that this equation is equivalent to these two

$$\mathsf{F}\ (h \times B) \cdot \mathit{bcl}_{\mathsf{F}} = \mathit{bcl}_{\mathsf{F}} \cdot (\mathsf{F}\ h \times B) \tag{7}$$

$$\mathsf{F}\ (A \times k) \cdot \mathit{bcl}_{\mathsf{F}} = \mathit{bcl}_{\mathsf{F}} \cdot (\mathsf{F}\ A \times k) \tag{8}$$

which capture naturality in one of the arguments. Thanks!

Teacher: Now, to define $\circledast$ it'll be handy to also have an operation that broadcasts to the right: $\mathit{bcr} : A \times \mathsf{F}\ B \to \mathsf{F}\ (A \times B)$. One can be defined in terms of the other: $\mathit{bcr} = \mathsf{F}\ \mathit{swap} \cdot \mathit{bcl} \cdot \mathit{swap}$, where $\mathit{swap} : A \times B \cong B \times A$ is the isomorphism that swaps a pair. We'll rename $\circledast$ to *com*; who wants to have a go?

Lisa: Let's see. We are given a pair of structures $\mathsf{F}\ A \times \mathsf{G}\ B$. The only thing we can really do is to broadcast $\mathsf{G}\ B$ over $\mathsf{F}\ A$, which gives $\mathsf{F}\ (A \times \mathsf{G}\ B)$, and then broadcast A over $\mathsf{G}\ B$, within F. So,

$$\begin{aligned}&\mathit{com}_{\mathsf{F},\mathsf{G}} \ :\ \mathsf{F}\ A \times \mathsf{G}\ B \to (\mathsf{F} \circ \mathsf{G})\ (A \times B)\\&\mathit{com}_{\mathsf{F},\mathsf{G}} = \mathsf{F}\ \mathit{bcr}_{\mathsf{G}} \cdot \mathit{bcl}_{\mathsf{F}}\end{aligned} \tag{9}$$

I've added F and G as subscripts as a substitute for Haskell's overloading. By the way, here we're making use of the fact that the broadcasts are natural. Furthermore, $\mathit{com}_{\mathsf{F},\mathsf{G}}$ inherits naturality: it's natural in A and B, as well.

Harry: Why can't we do it the other way round? First broadcast $\mathsf{F}\ A$ over $\mathsf{G}\ B \ldots$ No, then the functors come out in the wrong order.

Teacher: For completeness, here's the categorical definition of *unit*—recall that 1 is the counterpart of Haskell's unit type $()$.

$$\begin{aligned}&\mathit{unit} \ :\ 1 \to \mathsf{Id}\ 1\\&\mathit{unit} = \mathit{id}_1\end{aligned}$$

Lisa: Comparing *nil* and $\lozenge$ with *unit* and $\circledast$, aka *com*, it seems that we've replaced the *String* monoid by a monoid on the level of objects: $\times$ is associative with 1 as its neutral element.

Teacher: Yes, but we have to be careful to pick the right notion of equality: $(A \times B) \times C$ is not the *same* object as $A \times (B \times C)$, rather the two objects are isomorphic. We have

$$\lambda : 1 \times A \cong A \tag{10}$$

$$\rho : A \times 1 \cong A \tag{11}$$

$$\alpha : (A \times B) \times C \cong A \times (B \times C) \tag{12}$$

All of these have to be natural isomorphisms. Sorry for the Greek letters, but they're all standard: lambda cancels the unit of the left, rho on the right, and alpha is the associative law.

Lisa: So *com* is only associative modulo these isomorphisms?

Teacher: Correct. We have to add the isomorphisms at the appropriate places, but it's only slightly inconvenient. Here are the desired properties of *unit* and *com*: the unit and associative laws.

$$\mathsf{F}\,\lambda \cdot com_{\mathsf{Id},\mathsf{F}} \cdot (unit \times id) = \lambda \tag{13}$$

$$\mathsf{F}\,\rho \cdot com_{\mathsf{F},\mathsf{Id}} \cdot (id \times unit) = \rho \tag{14}$$

$$\begin{aligned}(\mathsf{F}\circ\mathsf{G}\circ\mathsf{H})\;\alpha \cdot com_{\mathsf{F}\circ\mathsf{G},\mathsf{H}} \cdot (com_{\mathsf{F},\mathsf{G}} \times id)\\ = com_{\mathsf{F},\mathsf{G}\circ\mathsf{H}} \cdot (id \times com_{\mathsf{G},\mathsf{H}}) \cdot \alpha\end{aligned} \tag{15}$$

If the isomorphisms λ, ρ and α were identities, then we'd simply write $com_{\mathsf{Id},\mathsf{F}} \cdot (unit \times id) = id$, and so forth.

Harry: These laws look a lot more daunting than the laws of the *String* monoid: $"" +\!\!+\, s = s$, $s +\!\!+ "" = s$ and $(s +\!\!+ t) +\!\!+ u = s +\!\!+ (t +\!\!+ u)$.

Teacher: It's possibly helpful to draw the corresponding diagrams.

$$\begin{array}{ccc} 1\times\mathsf{F}\,A & \xRightarrow{\lambda} & \mathsf{F}\,A \\ {\scriptstyle unit\times id}\downarrow & & \Vert{\scriptstyle \mathsf{F}\,\lambda} \\ \mathsf{Id}\;1\times\mathsf{F}\,A & \xrightarrow[com_{\mathsf{Id},\mathsf{F}}]{} & (\mathsf{Id}\circ\mathsf{F})\;(1\times A) \end{array} \qquad \begin{array}{ccc} \mathsf{F}\,A\times 1 & \xRightarrow{\rho} & \mathsf{F}\,A \\ {\scriptstyle id\times unit}\downarrow & & \Vert{\scriptstyle \mathsf{F}\,\rho} \\ \mathsf{F}\,A\times\mathsf{Id}\;1 & \xrightarrow[com_{\mathsf{F},\mathsf{Id}}]{} & (\mathsf{F}\circ\mathsf{Id})\;(A\times 1) \end{array}$$

$$\begin{array}{ccc} (\mathsf{F}\,A\times\mathsf{G}\,B)\times\mathsf{H}\,C & \xRightarrow{\alpha} & \mathsf{F}\,A\times(\mathsf{G}\,B\times\mathsf{H}\,C) \\ {\scriptstyle com_{\mathsf{F},\mathsf{G}}\times id}\downarrow & & \downarrow{\scriptstyle id\times com_{\mathsf{G},\mathsf{H}}} \\ ((\mathsf{F}\circ\mathsf{G})\;(A\times B))\times\mathsf{H}\,C & & \mathsf{F}\,A\times((\mathsf{G}\circ\mathsf{H})\;(B\times C)) \\ {\scriptstyle com_{\mathsf{F}\circ\mathsf{G},\mathsf{H}}}\downarrow & & \downarrow{\scriptstyle com_{\mathsf{F},\mathsf{G}\circ\mathsf{H}}} \\ (\mathsf{F}\circ\mathsf{G}\circ\mathsf{H})\;((A\times B)\times C) & \xRightarrow[(\mathsf{F}\circ\mathsf{G}\circ\mathsf{H})\;\alpha]{} & (\mathsf{F}\circ\mathsf{G}\circ\mathsf{H})\;(A\times(B\times C)) \end{array}$$

I always find the type information invaluable. By the way, the double lines in the diagram indicate isomorphism, not equality. I wanted to avoid drawing long twiddles $\sim$.

Harry: In a sense, these conditions are similar to the coherence condition for bifunctors. There are two ways of turning three structures $(\mathsf{F}\,A\times\mathsf{G}\,B)\times\mathsf{H}\,C$ into the nested structure $(\mathsf{F}\circ\mathsf{G}\circ\mathsf{H})\;(A\times(B\times C))$. The associative law expresses that both paths lead to the same result.

Teacher: That's a very insightful remark. Category theory has been characterised as coherently constructive lattice theory [2]. So yes, the laws can be seen as coherence requirements. Let's have a crack at the proofs, shall we?

Lisa: *Hmm*, it seems to me that we are missing some assumptions: coherence conditions for *bcl* and *bcr*.

Teacher: Have a go.

Lisa: I would expect stripped-down versions of the unit laws and the associative law (15). Since $\mathsf{F}\,A\times 1\cong\mathsf{F}\,A$, broadcasting the unit type should be a no-op. Since $(\mathsf{F}\,A\times B)\times C\cong\mathsf{F}\,A\times(B\times C)$, a nested broadcast of first B and then C, should be equivalent to a single broadcast of $B\times C$.

$$\mathsf{F}\,\rho \cdot bcl_{\mathsf{F}} = \rho \tag{16}$$

$$\mathsf{F}\,\alpha \cdot bcl_{\mathsf{F}} \cdot (bcl_{\mathsf{F}} \times id) = bcl_{\mathsf{F}} \cdot \alpha \tag{17}$$

$$\begin{array}{ccc} (\mathsf{F}\,A\times B)\times C & \xRightarrow{\alpha} & \mathsf{F}\,A\times(B\times C) \\ {\scriptstyle bcl_{\mathsf{F}}\times id}\downarrow & & \downarrow{\scriptstyle bcl_{\mathsf{F}}} \\ \mathsf{F}\,(A\times B)\times C & & \\ {\scriptstyle bcl_{\mathsf{F}}}\downarrow & & \\ \mathsf{F}\,((A\times B)\times C) & \xRightarrow[\mathsf{F}\,\alpha]{} & \mathsf{F}\,(A\times(B\times C)) \end{array} \qquad \begin{array}{ccc} \mathsf{F}\,A\times 1 & \xrightarrow{\rho} & \mathsf{F}\,A \\ {\scriptstyle bcl_{\mathsf{F}}}\downarrow & \nearrow{\scriptstyle \mathsf{F}\,\rho} & \\ \mathsf{F}\,(A\times 1) & & \end{array}$$

Harry: And for *bcr* we swap the laws around.

$$\mathsf{F}\,\lambda \cdot bcr_{\mathsf{F}} = \lambda \tag{18}$$

$$\mathsf{F}\,\alpha \cdot bcr_{\mathsf{F}} = bcr_{\mathsf{F}} \cdot (id \times bcr_{\mathsf{F}}) \cdot \alpha \tag{19}$$

$$\begin{array}{ccccc} & {\scriptstyle\lambda}\swarrow & 1\times\mathsf{F}\,A & & \\ \mathsf{F}\,A & & \downarrow{\scriptstyle bcr_{\mathsf{F}}} & & \\ & {\scriptstyle\mathsf{F}\,\lambda}\nwarrow & \mathsf{F}\,(1\times A) & & \end{array} \qquad \begin{array}{ccc} (A\times B)\times\mathsf{F}\,C & \xRightarrow{\alpha} & A\times(B\times\mathsf{F}\,C) \\ {\scriptstyle bcr_{\mathsf{F}}}\downarrow & & \downarrow{\scriptstyle id\times bcr_{\mathsf{F}}} \\ & & A\times\mathsf{F}\,(B\times C) \\ & & \downarrow{\scriptstyle bcr_{\mathsf{F}}} \\ \mathsf{F}\,((A\times B)\times C) & \xRightarrow[\mathsf{F}\,\alpha]{} & \mathsf{F}\,(A\times(B\times C)) \end{array}$$

Teacher: Well done. It will also be convenient to have a 'mixed' associative law.

$$\mathsf{F}\,\alpha \cdot bcl_{\mathsf{F}} \cdot (bcr_{\mathsf{F}} \times id) = bcr_{\mathsf{F}} \cdot (id \times bcl_{\mathsf{F}}) \cdot \alpha \tag{20}$$

$$\begin{array}{ccc} (A\times\mathsf{F}\,B)\times C & \xRightarrow{\alpha} & A\times(\mathsf{F}\,B\times C) \\ {\scriptstyle bcr_{\mathsf{F}}\times id}\downarrow & & \downarrow{\scriptstyle id\times bcl_{\mathsf{F}}} \\ \mathsf{F}\,(A\times B)\times C & & A\times\mathsf{F}\,(B\times C) \\ {\scriptstyle bcl_{\mathsf{F}}}\downarrow & & \downarrow{\scriptstyle bcr_{\mathsf{F}}} \\ \mathsf{F}\,((A\times B)\times C) & \xRightarrow[\mathsf{F}\,\alpha]{} & \mathsf{F}\,(A\times(B\times C)) \end{array}$$

To summarise, the broadcast operations have to be natural and they have to satisfy unit and associative laws. The two operations are interdefinable. Likewise, the laws for *bcr* follow from the laws for *bcl* and vice versa. The 'mixed' law is a consequence of either.

Lisa: Going back to the original problem, shouldn't we check that the functors involved do indeed support broadcast?

Teacher: Absolutely. Just take my word for it that Reader A does—a precise argument would lead us astray. I'll leave it to you to check the identity functor and functor composition.

Lisa: Well, Id is straightforward, we pick the identity for $bcl_{\mathsf{Id}} : \mathsf{Id}\,A\times B\to\mathsf{Id}\,(A\times B)$. For composition, we need to assume that the to-be-composed functors have broadcasts, then we can define $bcl_{\mathsf{F}\circ\mathsf{G}}$ in terms of bcl_{F} and bcl_{G}.

$$bcl_{\mathsf{Id}} = id \tag{21}$$

$$bcl_{\mathsf{F}\circ\mathsf{G}} = \mathsf{F}\,bcl_{\mathsf{G}} \cdot bcl_{\mathsf{F}} \tag{22}$$

$$bcr_{\mathsf{Id}} = id \tag{23}$$

$$bcr_{\mathsf{F}\circ\mathsf{G}} = \mathsf{F}\,bcr_{\mathsf{G}} \cdot bcr_{\mathsf{F}} \tag{24}$$

Lambert: Interesting. We defined *com* in terms of *bcl* and *bcr*. The symmetric *com* seems to generalise the asymmetric broadcasts. Using (21) and (23), we can actually make this precise: $com_{\mathsf{F},\mathsf{Id}} = bcl_{\mathsf{F}}$ and $com_{\mathsf{Id},\mathsf{F}} = bcr_{\mathsf{F}}$. Furthermore, the unit laws for *com*, (13) and (14), are immediate consequences of the unit laws for the broadcasts, (16) and (18).

Lisa (*animated*)**:** Similarly, the three associative laws, (17), (19) and (20), are instances of the associative law for *com* (15)! We simply instantiate two out of three functors to the identity functor.

Teacher: That's a brilliant observation. There is indeed something more general going on. The broadcast bcl_{F} can be seen as a family of natural transformations, indexed by the functor F. Your observations suggest that these instances are not independent of each other, *bcl* is what is called a polytypic or datatype-generic operation. If you'd like to investigate this further, then I heartily recommend paper [5] from our reading list. (There *bcl* is identified as a special case of a more general operation called *zip*.)

Lambert (*sweating*)**:** I've just completed the proof of the associative law (15). It's somewhat longish, but mostly straightforward. It would have even been longer, if I'd spelled out every single application of the functor laws.

[*Proof reproduced in Figure 1 with Lambert's kind permission.*]

Harry: I'm so impressed; I couldn't have come up with this.

Lambert: Well, there is a clear strategy: we have to move the isomorphism α from the left to the right. The three associative laws are the only means to accomplish this. At some points we have to appeal to naturality to make progress.

Teacher: No need to despair, Harry. There is actually a very systematic approach. The associative law states that the two ways of transforming $(\mathsf{F}\,A \times \mathsf{G}\,B) \times \mathsf{H}\,C$ into $(\mathsf{F} \circ \mathsf{G} \circ \mathsf{H})\,(A \times (B \times C))$ using *com* are equivalent. However, using the primitive operations *bcl* and *bcr* there are actually a lot more possibilities. I've prepared a little commutative diagram that details the various ways.

[*The Teacher distributes the diagram reproduced in Figure 2.*]

Recall that a diagram commutes if all of its inner polygons commute. For instance, the hexagon labelled ③ is an instance of the mixed associative law (20). Or as another example, ⑤ is a naturality square (7), which amounts to saying that the two rewrites are independent of each other.

Lisa: It's actually not too hard to relate the commutative diagram to Lambert's proof. I've added references to some of the steps, the remaining ones only plug in definitions.

Teacher (*peeks at his watch*)**:** OK. I think it's time to wrap up. I'd like to use the remaining half an hour to put today's findings into perspective. I noticed that you've all fallen in love with the 'functor application' functor. [*Harry sighs.*] What we've just shown is that this functor is also a monoidal functor.

Harry: A what?

Teacher: A monoidal functor is one between monoidal categories. So what is a monoidal category? A category with a bit of extra structure, a monoid on the object level: an object N and a bifunctor $\oplus$ that is associative with N as its neutral element. Can someone provide an example?

Harry: Well, Haskell's ambient category with the unit 1 and the binary product $\times$?

Teacher: Almost, but not quite.

Lisa: The category $\mathscr{C}^{\mathscr{C}}$ with the identity functor Id and functor composition $\circ$. We know that functor composition is a functor, so this seems to work out.

Teacher: Excellent. This makes a *strict* monoidal category. Harry's example is what is called a *relaxed* monoidal category. In the latter case, we replace the equalities by isomorphisms, see (10)–(12). These isomorphisms have to be natural and they have to satisfy suitable coherence conditions (which I'm too tired to discuss). How would you define the notion of a monoidal functor?

Lisa: I would expect that a monoidal functor preserves the extra structure. If $(\mathscr{C}, N, \oplus)$ and $(\mathscr{D}, E, \otimes)$ are monoidal categories, then a monoidal functor is a functor $\mathsf{F} : \mathscr{C} \to \mathscr{D}$ such that

$$\mathsf{F}\,N \cong E$$
$$\mathsf{F}\,(A \oplus B) \cong \mathsf{F}\,A \otimes \mathsf{F}\,B$$

In line with the previous discussion I've opted for isomorphism rather than equality.

Teacher: This is what is called a *strong* monoidal functor. A *relaxed* monoidal functor replaces the isomorphisms by arrows.

$$E \to \mathsf{F}\,N$$
$$\mathsf{F}\,A \otimes \mathsf{F}\,B \to \mathsf{F}\,(A \oplus B)$$

The second arrow has to be natural in A and B. Furthermore, they have to satisfy unit laws and an associative law—this was exactly today's task (13)–(15).

Harry: So this gives us the warm fuzzy feeling that we were on the right track?

Teacher (*smiles*)**:** I guess so. It's always reassuring when you discover some known structure in some unknown territory.

Lisa: I'm not sure I can match up the types with *unit* and *com* ...

Teacher: Well, this is the truly intriguing part: the monoidal functor is the application functor, which is a bifunctor from the strict monoidal category $\mathscr{C}^{\mathscr{C}}$ and the relaxed monoidal category $\mathscr{C}$ to $\mathscr{C}$. (The product of two monoidal categories is again monoidal.) So the types of unit and com are

$$unit : \mathsf{Id} \star 1$$
$$com_{\mathsf{F},\mathsf{G}} : \mathsf{F} \star A \times \mathsf{G} \star B \to (\mathsf{F} \circ \mathsf{G}) \star (A \times B)$$

Lisa: *Hmm*, but then *com* has to be natural not only in A and B, but also in F and G?!?

Teacher: Yes, that's right: *com* is an example of a higher-order natural transformation. That said, we have to be slightly more careful with $\star$'s type.

Lisa: Yes, of course. The objects of $\mathscr{C}^{\mathscr{C}}$ can't be arbitrary functors, they have to support broadcasts.

Teacher: Well spotted. But this alone is not enough. We also have to require that the natural transformations respect the broadcasts. A natural transformation $\sigma : \mathsf{F} \dot{\to} \mathsf{G}$ is well-behaved in this sense if

$$\sigma\,(A \times B) \cdot bcl_{\mathsf{F}} = bcl_{\mathsf{G}} \cdot (\sigma\,A \times B) \tag{25}$$

$$\begin{array}{ccc} \mathsf{F}\,A \times B & \xrightarrow{\sigma\,A \times B} & \mathsf{G}\,A \times B \\ {\scriptstyle bcl_{\mathsf{F}}}\downarrow & & \downarrow{\scriptstyle bcl_{\mathsf{G}}} \\ \mathsf{F}\,(A \times B) & \xrightarrow[\sigma\,(A \times B)]{} & \mathsf{G}\,(A \times B) \end{array}$$

It doesn't matter whether we first broadcast and then change the structure, or the other way round.

So we have shown that functor application is a functor of type $\mathbf{BC}(\mathscr{C}) \times \mathscr{C} \to \mathscr{C}$, where $\mathbf{BC}(\mathscr{C})$ is the category of functors of type $\mathscr{C} \to \mathscr{C}$ that support broadcasts and natural transformations that respect broadcasts.

Lisa: *Hmm*, equation (25) looks a lot like a naturality property.

Teacher: Indeed. Looking at the equation from a different angle it expresses that bcl_{F} is natural in the functor F. If we combine the higher- and first-order naturality properties we obtain

$$\sigma \star (f \times g) \cdot bcl_{\mathsf{F}} = bcl_{\mathsf{G}} \cdot (\sigma \star f \times g) \tag{26}$$
$$\sigma \star (f \times g) \cdot bcr_{\mathsf{F}} = bcr_{\mathsf{G}} \cdot (f \times \sigma \star g) \tag{27}$$

Lambert: This is the missing link! We can now show that *com* is natural in both the functors and the objects. Let $\sigma : \mathsf{F} \dot{\to} \mathsf{H}$ and

$$
\begin{aligned}
& \mathsf{F}\,(\mathsf{G}\,(\mathsf{H}\,\alpha)) \cdot com_{\mathsf{F}\circ\mathsf{G},\mathsf{H}} \cdot (com_{\mathsf{F},\mathsf{G}} \times id) \\
= \quad & \{\ \text{definition of } com\ (9)\ \} \\
& \mathsf{F}\,(\mathsf{G}\,(\mathsf{H}\,\alpha)) \cdot \mathsf{F}\,(\mathsf{G}\,bcr_{\mathsf{H}}) \cdot bcl_{\mathsf{F}\circ\mathsf{G}} \cdot (\mathsf{F}\,bcr_{\mathsf{G}} \cdot bcl_{\mathsf{F}} \times id) \\
= \quad & \{\ \text{definition of } bcl_{\mathsf{F}\circ\mathsf{G}}\ (22)\ \} \\
& \mathsf{F}\,(\mathsf{G}\,(\mathsf{H}\,\alpha)) \cdot \mathsf{F}\,(\mathsf{G}\,bcr_{\mathsf{H}}) \cdot \mathsf{F}\,bcl_{\mathsf{G}} \cdot bcl_{\mathsf{F}} \cdot (\mathsf{F}\,bcr_{\mathsf{G}} \cdot bcl_{\mathsf{F}} \times id) \\
= \quad & \{\ ①\ bcr_{\mathsf{H}} \text{ associative } (19)\ \} \\
& \mathsf{F}\,(\mathsf{G}\,(bcr_{\mathsf{H}} \cdot (id \times bcr_{\mathsf{H}}) \cdot \alpha)) \cdot \mathsf{F}\,bcl_{\mathsf{G}} \cdot bcl_{\mathsf{F}} \cdot (\mathsf{F}\,bcr_{\mathsf{G}} \cdot bcl_{\mathsf{F}} \times id) \\
= \quad & \{\ \times \text{ bifunctor}\ \} \\
& \mathsf{F}\,(\mathsf{G}\,(bcr_{\mathsf{H}} \cdot (id \times bcr_{\mathsf{H}}) \cdot \alpha)) \cdot \mathsf{F}\,bcl_{\mathsf{G}} \cdot bcl_{\mathsf{F}} \cdot (\mathsf{F}\,bcr_{\mathsf{G}} \times id) \cdot (bcl_{\mathsf{F}} \times id) \\
= \quad & \{\ ②\ bcl_{\mathsf{F}} \text{ natural in } A\ (8)\ \} \\
& \mathsf{F}\,(\mathsf{G}\,(bcr_{\mathsf{H}} \cdot (id \times bcr_{\mathsf{H}}) \cdot \alpha)) \cdot \mathsf{F}\,bcl_{\mathsf{G}} \cdot \mathsf{F}\,(bcr_{\mathsf{G}} \times id) \cdot bcl_{\mathsf{F}} \cdot (bcl_{\mathsf{F}} \times id) \\
= \quad & \{\ ③\ bcl_{\mathsf{G}}/bcr_{\mathsf{G}} \text{ associative } (20)\ \} \\
& \mathsf{F}\,(\mathsf{G}\,(bcr_{\mathsf{H}} \cdot (id \times bcr_{\mathsf{H}}))) \cdot \mathsf{F}\,(bcr_{\mathsf{G}} \cdot (id \times bcl_{\mathsf{G}}) \cdot \alpha) \cdot bcl_{\mathsf{F}} \cdot (bcl_{\mathsf{F}} \times id) \\
= \quad & \{\ ④\ bcl_{\mathsf{F}} \text{ associative } (17)\ \} \\
& \mathsf{F}\,(\mathsf{G}\,(bcr_{\mathsf{H}} \cdot (id \times bcr_{\mathsf{H}}))) \cdot \mathsf{F}\,(bcr_{\mathsf{G}} \cdot (id \times bcl_{\mathsf{G}})) \cdot bcl_{\mathsf{F}} \cdot \alpha \\
= \quad & \{\ ⑤\ bcl_{\mathsf{F}} \text{ natural in } B\ (7)\ \} \\
& \mathsf{F}\,(\mathsf{G}\,(bcr_{\mathsf{H}} \cdot (id \times bcr_{\mathsf{H}}))) \cdot \mathsf{F}\,bcr_{\mathsf{G}} \cdot bcl_{\mathsf{F}} \cdot (id \times bcl_{\mathsf{G}}) \cdot \alpha \\
= \quad & \{\ ⑥\ bcr_{\mathsf{G}} \text{ natural in } B\ \} \\
& \mathsf{F}\,(\mathsf{G}\,bcr_{\mathsf{H}} \cdot bcr_{\mathsf{G}} \cdot (id \times \mathsf{G}\,bcr_{\mathsf{H}})) \cdot bcl_{\mathsf{F}} \cdot (id \times bcl_{\mathsf{G}}) \cdot \alpha \\
= \quad & \{\ ⑦\ bcl_{\mathsf{F}} \text{ natural in } B\ (7)\ \} \\
& \mathsf{F}\,(\mathsf{G}\,bcr_{\mathsf{H}} \cdot bcr_{\mathsf{G}}) \cdot bcl_{\mathsf{F}} \cdot (id \times \mathsf{G}\,bcr_{\mathsf{H}}) \cdot (id \times bcl_{\mathsf{G}}) \cdot \alpha \\
= \quad & \{\ \times \text{ bifunctor}\ \} \\
& \mathsf{F}\,(\mathsf{G}\,bcr_{\mathsf{H}} \cdot bcr_{\mathsf{G}}) \cdot bcl_{\mathsf{F}} \cdot (id \times \mathsf{G}\,bcr_{\mathsf{H}} \cdot bcl_{\mathsf{G}}) \cdot \alpha \\
= \quad & \{\ \text{definition of } bcr_{\mathsf{G}\circ\mathsf{H}}\ (24)\ \} \\
& \mathsf{F}\,bcr_{\mathsf{G}\circ\mathsf{H}} \cdot bcl_{\mathsf{F}} \cdot (id \times \mathsf{G}\,bcr_{\mathsf{H}} \cdot bcl_{\mathsf{G}}) \cdot \alpha \\
= \quad & \{\ \text{definition of } com\ (9)\ \} \\
& com_{\mathsf{F},\mathsf{G}\circ\mathsf{H}} \cdot (id \times com_{\mathsf{G},\mathsf{H}}) \cdot \alpha
\end{aligned}
$$

Figure 1. Calculational proof of the associative law (15)

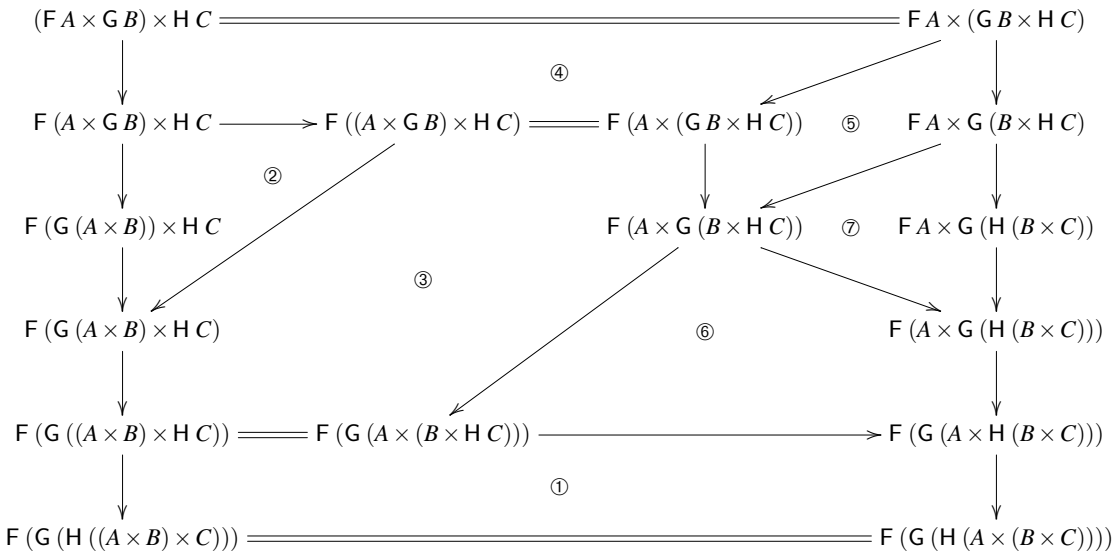


Figure 2. Commuting diagram proving the associative law (15)

$\tau : \mathsf{G} \dot{\to} \mathsf{K}$, then

$$
\begin{aligned}
& com_{\mathsf{H},\mathsf{K}} \cdot (\sigma \star f \times \tau \star g) \\
= \quad & \{ \text{ definition of } com_{\mathsf{H},\mathsf{K}} \ (9) \ \} \\
& \mathsf{H}\, bcr_{\mathsf{K}} \cdot bcl_{\mathsf{H}} \cdot (\sigma \star f \times \tau \star g) \\
= \quad & \{ \ bcl \text{ is higher-order natural } (26) \ \} \\
& \mathsf{H}\, bcr_{\mathsf{K}} \cdot \sigma \star (f \times \tau \star g) \cdot bcl_{\mathsf{F}} \\
= \quad & \{ \ \star \text{ bifunctor } \} \\
& \sigma \star (bcr_{\mathsf{K}} \cdot (f \times \tau \star g)) \cdot bcl_{\mathsf{F}} \\
= \quad & \{ \ bcr \text{ is higher-order natural } (27) \ \} \\
& \sigma \star (\tau \star (f \times g) \cdot bcr_{\mathsf{G}}) \cdot bcl_{\mathsf{F}} \\
= \quad & \{ \ \star \text{ bifunctor } \} \\
& \sigma \star (\tau \star (f \times g)) \cdot \mathsf{F}\, bcr_{\mathsf{G}} \cdot bcl_{\mathsf{F}} \\
= \quad & \{ \text{ definition of } com_{\mathsf{F},\mathsf{G}} \ (9) \ \} \\
& \sigma \star (\tau \star (f \times g)) \cdot com_{\mathsf{F},\mathsf{G}} \\
= \quad & \{ \ \sigma \star (\tau \star h) = (\sigma \circ \tau) \star h, \text{ see Saturday } \} \\
& (\sigma \circ \tau) \star (f \times g) \cdot com_{\mathsf{F},\mathsf{G}}
\end{aligned}
$$

Teacher: Excellent. I think we're done. As a final remark: monoidal functors have become rather fashionable lately in the Haskell community, where they're called idioms or applicative functors [8]. Like monads, they are useful for adding structure and genericity to your programs [1, 7, 10].

Well, I hope you've enjoyed the tutorials this week as much as I have—even though we are badly overrunning today.

Harry: Definitely. Today's material was somewhat over my head—well, I asked for it. Anyway, I'm certainly going to remember that there's a lot more to functors than what the Haskell *Functor* type class suggests.

Lisa: *Oh* Harry, you ain't seen nothin' yet. I've been reading ahead and I saw these things called *Kan extensions*; apparently *everything* is a Kan extension!

Acknowledgments

The authors would like to thank Steve McKeever for many helpful comments on earlier drafts of this paper. Daniel is funded by a DTA Studentship from the Engineering and Physical Sciences Research Council (EPSRC). This work is partially supported by the EPSRC grant EP/J010995/1.

Reading List

[1] A. I. Baars, A. Löh, and S. D. Swierstra. Functional pearl: Parsing permutation phrases. *Journal of Functional Programming*, 14(6):635–646, 2004. doi: 10.1017/S0956796804005143.

[2] R. Backhouse, M. Bijsterveld, R. van Geldrop, and J. van der Woude. Category Theory As Coherently Constructive Lattice Theory, 2003. Working Document, available from `http://www.cs.nott.ac.uk/~rcb/MPC/CatTheory.ps.gz`.

[3] J. Gibbons. Datatype-Generic Programming. In R. Backhouse, J. Gibbons, R. Hinze, and J. Jeuring, editors, *Datatype-Generic Programming*, volume 4719 of *Lecture Notes in Computer Science*, pages 1–71. Springer, 2007. doi: 10.1007/978-3-540-76786-2_1.

[4] R. Hinze. Functional Pearl: Formatting: a class act. *Journal of Functional Programming*, 13(5):935–944, 2003. doi: 10.1017/S0956796802004367.

[5] P. Hoogendijk and R. Backhouse. When do datatypes commute? In E. Moggi and G. Rosolini, editors, *Proceedings of the 7th International Conference on Category Theory and Computer Science*, volume 1290 of *Lecture Notes in Computer Science*, pages 242–260. Springer, 1997. doi: 10.1007/BFb002699.

[6] P. Johann and N. Ghani. Initial Algebra Semantics Is Enough! In S. Della Rocca, editor, *Typed Lambda Calculi and Applications*, volume 4583 of *Lecture Notes in Computer Science*, pages 207–222. Springer, 2007. doi: 10.1007/978-3-540-73228-0_16.

[7] K. Matlage and A. Gill. Every Animation Should Have a Beginning, a Middle, and an End. In R. Page, Z. Horváth, and V. Zsók, editors, *Trends in Functional Programming*, volume 6546 of *Lecture Notes in Computer Science*, pages 150–165. Springer, 2011. doi: 10.1007/978-3-642-22941-1_10.

[8] C. McBride and R. Paterson. Functional Pearl: Applicative programming with effects. *Journal of Functional Programming*, 18(1):1–13, 2008. doi: 10.1017/S0956796807006326.

[9] M. Neubauer and P. Thiemann. Type classes with more higher-order polymorphism. In *Proceedings of the 7th International Conference on Functional programming*, ICFP '02, pages 179–190. ACM, September 2002. doi: 10.1145/581478.581496.

[10] S. Swierstra and L. Duponcheel. Deterministic, error-correcting combinator parsers. In J. Launchbury, E. Meijer, and T. Sheard, editors, *Advanced Functional Programming*, volume 1129 of *Lecture Notes in Computer Science*, pages 184–207. Springer, 1996. doi: 10.1007/3-540-61628-4_7.