$$
\begin{array}{lll}
\text{axioms :} & \dfrac{}{\Gamma \vdash \mathcal{CH}(\mathbb{1})} \quad \text{(use unit)} & \dfrac{}{\Gamma, \alpha \vdash \alpha} \quad \text{(use arg)} \\
\text{derivation rules :} & \dfrac{\Gamma, \alpha \vdash \beta}{\Gamma \vdash \alpha \Rightarrow \beta} \quad \text{(create function)} & \\
& \dfrac{\Gamma \vdash \alpha \qquad \Gamma \vdash \alpha \Rightarrow \beta}{\Gamma \vdash \beta} \quad \text{(use function)} & \\
& \dfrac{\Gamma \vdash \alpha \qquad \Gamma \vdash \beta}{\Gamma \vdash \alpha \wedge \beta} \quad \text{(create tuple)} & \\
& \dfrac{\Gamma \vdash \alpha \wedge \beta}{\Gamma \vdash \alpha} \quad \text{(use tuple-1)} & \dfrac{\Gamma \vdash \alpha \wedge \beta}{\Gamma \vdash \beta} \quad \text{(use tuple-2)} \\
& \dfrac{\Gamma \vdash \alpha}{\Gamma \vdash \alpha \vee \beta} \quad \text{(create }\texttt{Left}\text{)} & \dfrac{\Gamma \vdash \beta}{\Gamma \vdash \alpha \vee \beta} \quad \text{(create }\texttt{Right}\text{)} \\
& \dfrac{\Gamma \vdash \alpha \vee \beta \qquad \Gamma, \alpha \vdash \gamma \qquad \Gamma, \beta \vdash \gamma}{\Gamma \vdash \gamma} \quad \text{(use }\texttt{Either}\text{)} &
\end{array}
$$

Table 5.4: Proof rules for the constructive logic.

5.2.4 Example: Proving a $\mathcal{CH}$-proposition and deriving code

The task is to implement a fully parametric function

```
def f[A, B]: ((A => A) => B) => B = ???
```

Implementing this function is the same as being able to compute a value of type F, where F is defined as

$$F \triangleq \forall(A, B).\, ((A \rightarrow A) \rightarrow B) \rightarrow B \quad .$$

Since the type parameters A and B are arbitrary, the body of the fully parametric function `f` cannot use any previously defined values of types A or B. So, the task is formulated as computing a value of type F with *no* previously defined values. This is written as the sequent $\Gamma \vdash \mathcal{CH}(F)$, where the set Γ of premises is empty, $\Gamma = \emptyset$. Rewriting this sequent using the rules of Table 5.1, we get

$$\forall(\alpha, \beta).\ \emptyset \vdash ((\alpha \Rightarrow \alpha) \Rightarrow \beta) \Rightarrow \beta \quad , \tag{5.8}$$

where we denoted $\alpha \triangleq \mathcal{CH}(A)$ and $\beta \triangleq \mathcal{CH}(B)$.

The next step is to prove the sequent (5.8) using the logic proof rules of Section 5.2.3. For brevity, we will omit the quantifier $\forall(\alpha, \beta)$ since it will be present in front of every sequent.

Begin by looking for a proof rule whose "denominator" has a sequent similar to Eq. (5.8), i.e., has an implication $(p \Rightarrow q)$ in the goal. We have only one rule that can prove a sequent of the form $\Gamma \vdash (p \Rightarrow q)$; this is the rule "create function". That rule requires us to already have a proof of the sequent $(\Gamma, p) \vdash q$. So, we use this rule with $\Gamma = \emptyset$, and we set $p \triangleq (\alpha \Rightarrow \alpha) \Rightarrow \beta$ and $q \triangleq \beta$:

$$\frac{(\alpha \Rightarrow \alpha) \Rightarrow \beta \vdash \beta}{\emptyset \vdash ((\alpha \Rightarrow \alpha) \Rightarrow \beta) \Rightarrow \beta} \quad .$$

We now need to prove the sequent $(\alpha \Rightarrow \alpha) \Rightarrow \beta) \vdash \beta$, which we can write as $\Gamma_1 \vdash \beta$ where $\Gamma_1 \triangleq [(\alpha \Rightarrow \alpha) \Rightarrow \beta]$ denotes the set containing the single premise $(\alpha \Rightarrow \alpha) \Rightarrow \beta$.

There are no proof rules that derive a sequent with an explicit premise of the form of an implication $p \Rightarrow q$. However, we have a rule called "use function" that derives a sequent by assuming another sequent containing an implication. We would be able to use that rule,

$$\frac{\Gamma_1 \vdash \alpha \Rightarrow \alpha \qquad \Gamma_1 \vdash (\alpha \Rightarrow \alpha) \Rightarrow \beta}{\Gamma_1 \vdash \beta} \quad ,$$

if we could prove the two sequents $\Gamma_1 \vdash \alpha \Rightarrow \alpha$ and $\Gamma_1 \vdash (\alpha \Rightarrow \alpha) \Rightarrow \beta$. To prove these sequents, note that the rule "create function" applies to $\Gamma_1 \vdash \alpha \Rightarrow \alpha$ like this,

$$\frac{\Gamma_1, \alpha \vdash \alpha}{\Gamma_1 \vdash \alpha \Rightarrow \alpha} \quad .$$

(b) The type constructor $\text{Data}^{A,B}$ has *two* type parameters, and so we need to answer the question separately for each of them. Write the Scala type definition as

```
type Data[A, B] = (Either[A, B], (A => Int) => B)
```

Begin with the type parameter A and notice that a value of type $\text{Data}^{A,B}$ possibly contains a value of type A within `Either[A, B]`. In other words, A is "wrapped", i.e., it is in a covariant position within the first part of the tuple. It remains to check the second part of the tuple, which is a higher-order function of type $(A \to \text{Int}) \to B$. That function consumes a function of type $A \to \text{Int}$, which in turn consumes a value of type A. Consumers of A are contravariant in A, but it turns out that a "consumer of a consumer of A" is *covariant* in A. So we expect to be able to implement `fmap` that applies to the type parameter A of $\text{Data}^{A,B}$. Renaming the type parameter B to Z for clarity, we write the type signature for `fmap` like this,

$$\text{fmap}^{A,C,Z} : (A \to C) \to (A + Z) \times ((A \to \text{Int}) \to Z) \to (C + Z) \times ((C \to \text{Int}) \to Z) \quad .$$

We need to transform each part of the tuple separately. Transforming $A + Z$ into $C + Z$ is straightforward via the function

$$\begin{array}{c||cc} & C & Z \\ \hline A & f & \mathbb{0} \\ Z & \mathbb{0} & \text{id} \end{array} \quad .$$

This code notation corresponds to the following Scala code:

```
{
  case Left(x)    => Left(f(x))
  case Right(z)   => Right(z)
}
```

To derive code transforming $(A \to \text{Int}) \to Z$ into $(C \to \text{Int}) \to Z$, we use typed holes:

$$\begin{aligned}
& f^{:A\to C} \to g^{:(A\to\text{Int})\to Z} \to \underline{???^{:(C\to\text{Int})\to Z}} \\
\text{nameless function :}\quad & = f^{:A\to C} \to g^{:(A\to\text{Int})\to Z} \to p^{:C\to\text{Int}} \to \underline{???^{:Z}} \\
\text{get a } Z \text{ by applying } g\text{ :}\quad & = f^{:A\to C} \to g^{:(A\to\text{Int})\to Z} \to p^{:C\to\text{Int}} \to g(\underline{???^{:A\to\text{Int}}}) \\
\text{nameless function :}\quad & = f^{:A\to C} \to g^{:(A\to\text{Int})\to Z} \to p^{:C\to\text{Int}} \to g(a^{:A} \to \underline{???^{:\text{Int}}}) \\
\text{get an Int by applying } p\text{ :}\quad & = f^{:A\to C} \to g^{:(A\to\text{Int})\to Z} \to p^{:C\to\text{Int}} \to g(a^{:A} \to p(\underline{???^{:C}})) \\
\text{get a } C \text{ by applying } f\text{ :}\quad & = f^{:A\to C} \to g^{:(A\to\text{Int})\to Z} \to p^{:C\to\text{Int}} \to g(a^{:A} \to p(f(\underline{???^{:A}}))) \\
\text{use argument } a^{:A}\text{ :}\quad & = f \to g \to p \to g(a \to p(f(a))) \quad .
\end{aligned}$$

In the resulting Scala code for `fmap`, we write out some types for clarity:

```
def fmapA[A, Z, C](f: A => C): Data[A, Z] => Data[C, Z] = {
  case (e: Either[A, Z], g: ((A => Int) => Z)) =>
    val newE: Either[C, Z] = e match {
      case Left(x)    => Left(f(x))
      case Right(z)   => Right(z)
    }
    val newG: (C => Int) => Z = { p => g(a => p(f(a))) }
    (newE, newG) // This has type Data[C, Z].
}
```

This suggests that $\text{Data}^{A,Z}$ is covariant with respect to the type parameter A. The results of Section 6.2 will show rigorously that the functor laws hold for this implementation of `fmap`.

The analysis is simpler for the type parameter B because it is only used in covariant positions, never to the left of function arrows. So we expect $\text{Data}^{A,B}$ to be a functor with respect to B. Implementing the corresponding `fmap` is straightforward:

and contrafunctor methods for S ($\text{fmap}_{S^{A,\bullet}}$ and $\text{cmap}_{S^{\bullet,R}}$) are fully parametric. We omit the details since they are quite similar to what we saw in Section 6.2.2 for bifunctors.

If we define a type constructor $L^\bullet$ using the recursive "type equation"

$$L^A \triangleq S^{A,L^A} \triangleq (A \to \text{Int}) + L^A \times L^A \quad ,$$

we obtain a contrafunctor in the shape of a binary tree whose leaves are functions of type $A \to \text{Int}$. The next statement shows that recursive type equations of this kind always define contrafunctors.

Statement 6.2.4.3 If $S^{A,R}$ is a contrafunctor with respect to A and a functor with respect to R then the recursively defined type constructor C^A is a contrafunctor,

$$C^A \triangleq S^{A,C^A} \quad .$$

Given the functions $\texttt{cmap}_{S^{\bullet,R}}$ and $\texttt{fmap}_{S^{A,\bullet}}$ for S, we implement $\texttt{cmap}_C$ as

$$\text{cmap}_C(f^{:B\to A}) : C^A \to C^B \cong S^{A,C^A} \to S^{B,C^B} \quad ,$$
$$\text{cmap}_C(f^{:B\to A}) \triangleq \text{xmap}_S(f)(\overline{\text{cmap}_C}(f)) \quad .$$

The corresponding Scala code can be written as

```
final case class C[A](x: S[A, C[A]]) // The type constructor S[_, _] must be defined previously.

def xmap_S[A,B,Q,R](f: B => A)(g: Q => R): S[A, Q] => S[B, R] = ???          // Must be defined.

def cmap_C[A, B](f: B => A): C[A] => C[B] = { case C(x) =>
  val sbcb: S[B, C[B]] = xmap_S(f)(cmap_C(f))(x)                        // Recursive call to cmap_C.
  C(sbcb)              // Need to wrap the value of type S[B, C[B]] into the type constructor C.
}
```

Proof The code of `cmap` is recursive, and the recursive call is marked by an overline:

$$\text{cmap}_C(f) \triangleq f^{\downarrow C} \triangleq \text{xmap}_S(f)(\overline{\text{cmap}_C}(f)) \quad .$$

To verify the identity law:

$$\begin{aligned}
\text{expect to equal id :}\quad & \text{cmap}_C(\text{id}) = \text{xmap}_S(\text{id})(\underline{\overline{\text{cmap}_C}(\text{id})}) \\
\text{inductive assumption :}\quad & = \text{xmap}_S(\text{id})(\text{id}) \\
\text{identity law of } \text{xmap}_S\text{ :}\quad & = \text{id} \quad .
\end{aligned}$$

To verify the composition law:

$$\begin{aligned}
\text{expect to equal } (g^{\downarrow C} \,;\, f^{\downarrow C})\text{ :}\quad & (f^{:D\to B} \,;\, g^{:B\to A})^{\downarrow C} = \text{xmap}_S(f \,;\, g)(\underline{\overline{\text{cmap}_C}(f \,;\, g)}) \\
\text{inductive assumption :}\quad & = \text{xmap}_S(f \,;\, g)(\overline{\text{cmap}_C}(g) \,;\, \overline{\text{cmap}_C}(f)) \\
\text{composition law of } \text{xmap}_S\text{ :}\quad & = \text{xmap}_S(g)(\overline{\text{cmap}_C}(g)) \,;\, \text{xmap}_S(f)(\overline{\text{cmap}_C}(f)) \\
\text{definition of } {}^{\downarrow C}\text{ :}\quad & = g^{\downarrow C} \,;\, f^{\downarrow C} \quad .
\end{aligned}$$

6.2.5 Solved examples: How to recognize functors and contrafunctors

Sections 6.2.3 and 6.2.4 describe how functors and contrafunctors are built from other type expressions. We can see from Tables 6.2 and 6.4 that *every* one of the six basic type constructions (unit type, type parameters, product types, co-product types, function types, recursive types) gives either a new functor or a new contrafunctor. The six type constructions generate all exponential-polynomial types, including recursive ones. So, we should be able to decide whether any given exponential-polynomial type expression is a functor or a contrafunctor. The decision algorithm is based on the results shown in Tables 6.2 and 6.4:

Mathematical notation	Scala code
$x \to \sqrt{x^2+1}$	`x => math.sqrt(x*x + 1)`
list $[1, 2, ..., n]$	`(1 to n)`
list $[f(1), ..., f(n)]$	`(1 to n).map(k => f(k))`
$\sum_{k=1}^{n} k^2$	`(1 to n).map(k => k*k).sum`
$\prod_{k=1}^{n} f(k)$	`(1 to n).map(f).product`
$\forall k \in [1,...,n].\ p(k)$ holds	`(1 to n).forall(k => p(k))`
$\exists k \in [1,...,n].p(k)$ holds	`(1 to n).exists(k => p(k))`
$\sum_{k \in S \text{ such that } p(k) \text{ holds}} f(k)$	`s.filter(p).map(f).sum`

Table 1.1: Translating mathematics into code.

1.4.2 Transformation

Example 1.4.2.1 Given a list of lists, `s: List[List[Int]]`, select the inner lists of size at least 3. The result must be again of type `List[List[Int]]`.

Solution To "select the inner lists" means to compute a *new* list containing only the desired inner lists. We use `filter` on the outer list `s`. The predicate for the filter is a function that takes an inner list and returns `true` if the size of that list is at least 3. Write the predicate as a nameless function, `t => t.size >= 3`, where `t` is of type `List[Int]`:

```
def f(s: List[List[Int]]): List[List[Int]] = s.filter(t => t.size >= 3)

scala> f(List( List(1,2), List(1,2,3), List(1,2,3,4) ))
res0: List[List[Int]] = List(List(1, 2, 3), List(1, 2, 3, 4))
```

The Scala compiler deduces the type of `t` from the code; no other type would work since we apply `filter` to a *list of lists* of integers.

Example 1.4.2.2 Find all integers $k \in [1, 10]$ such that there are at least three different integers j, where $1 \le j \le k$, each j satisfying the condition $j^2 > 2k$.

Solution

```
scala> (1 to 10).toList.filter(k => (1 to k).filter(j => j*j > 2*k).size >= 3)
res0: List[Int] = List(6, 7, 8, 9, 10)
```

The argument of the outer `filter` is a nameless function that also uses a `filter`. The inner expression

```
(1 to k).filter(j => j*j > 2*k).size >= 3
```

(shown at left) computes the list of j's that satisfy the condition $j^2 > 2k$, and then compares the size of that list with 3. In this way, we impose the requirement that there should be at least 3 values of j. We can see how the Scala code closely follows the mathematical formulation of the task.

1.5 Summary

Functional programs are mathematical formulas translated into code. Table 1.1 shows how to implement some often used mathematical constructions in Scala.

What problems can one solve with this knowledge?

- Compute mathematical expressions involving sums, products, and quantifiers, based on integer ranges, such as $\sum_{k=1}^{n} f(k)$ etc.

Omitting the common sub-expressions, we find the remaining difference:

$$\text{liftOpt}_G(f')(\text{liftOpt}_G(f)(g)) \overset{?}{=} \text{liftOpt}_G\big(f \,;\, \text{flm}_{\text{Opt}}(f')\big)(g) \quad .$$

This is equivalent to liftOpt_G's composition law applied to the function g,

$$g \triangleright \text{liftOpt}_G(f) \,;\, \text{liftOpt}_G(f') = g \triangleright \text{liftOpt}_G(\underline{f \diamond_{\text{Opt}} f'}) = g \triangleright \text{liftOpt}_G\big(f \,;\, \text{flm}_{\text{Opt}}(f')\big) \quad .$$

Since the composition law of liftOpt_G is assumed to hold, we have finished the proof of Eq. (9.39).

The construction in Statement 9.2.4.4 implements a special kind of filtering where the value $a^{:A}$ in the pair of type $A \times G^A$ needs to pass the filter for any data to remain in the functor after filtering. We can use the same construction repeatedly with $G^\bullet \triangleq \mathbb{1}$ and obtain the type

$$L_n^A \triangleq \underbrace{\mathbb{1} + A \times (\mathbb{1} + A \times (\mathbb{1} + ... \times (\mathbb{1} + A \times \mathbb{1})))}_{\text{parameter } A \text{ is used } n \text{ times}} \quad ,$$

which is equivalent to a list of up to n elements. The construction defines a filtering operation for $L_n^\bullet$ that will delete any data beyond the first value of type A that does fails the predicate. It is clear that this filtering operation implements the standard `takeWhile` method defined on sequences. So, `takeWhile` is a lawful filtering operation (see Example 9.1.4.3 where it was used).

We can also generalize the construction of Statement 9.2.4.4 to the functor

$$F^A \triangleq \mathbb{1} + \underbrace{A \times A \times ... \times A}_{n \text{ times}} \times G^A \quad .$$

We implement the filtering operation with the requirement that *all* n values of type A in the tuple $A \times A \times ... \times A \times G^A$ must pass the filtering predicate, or else F^A becomes empty. Example 9.1.4.2 shows how such filtering operations may be used in practice.

Function types As we have seen in Chapter 6 (Statement 6.2.3.5), functors involving a function type, such as $F^A \triangleq G^A \to H^A$, require $G^\bullet$ to be a *contrafunctor* rather than a functor. It turns out that the functor $G^A \to H^A$ is filterable only if the contrafunctor $G^\bullet$ has certain properties (Eqs. (9.50)–(9.51) below) similar to properties of filterable functors. We will call such contrafunctors **filterable**.

To motivate the definition of filterable contrafunctors, consider the operation `liftOpt` for F:

$$\text{liftOpt}_F(f^{:A\to\mathbb{1}+B}) : (G^A \to H^A) \to G^B \to H^B \quad , \qquad \text{liftOpt}_F(f) = p^{:G^A\to H^A} \to g^{:G^B} \to ???^{:H^B} \quad .$$

Assume that H is filterable, so that we have the function $\text{liftOpt}_H(f) : H^A \to H^B$. We will fill the typed hole $???^{:H^B}$ if we somehow get a value of type H^A; that is only possible if we apply $p^{:G^A\to H^A}$,

$$\text{liftOpt}_F(f) = p^{:G^A\to H^A} \to g^{:G^B} \to \text{liftOpt}_H(f)(p(???^{:G^A})) \quad .$$

The only way to proceed is to have a function $G^B \to G^A$. We cannot obtain such a function by lifting f to the contrafunctor G: that gives $f^{\downarrow G} : G^{\mathbb{1}+B} \to G^A$. So, we need to require having a function

$$\text{liftOpt}_G(f^{:A\to\mathbb{1}+B}) : G^B \to G^A \quad . \tag{9.48}$$

This function is analogous to `liftOpt` for functors, except for the reverse direction of transformation ($G^B \to G^A$ instead of $G^A \to G^B$). We can now complete the implementation of liftOpt_F:

$$\begin{aligned}
& \text{liftOpt}_F(f^{:A\to\mathbb{1}+B}) \triangleq p^{:G^A\to H^A} \to g^{:G^B} \to \underline{\text{liftOpt}_H(f)\big(p(\text{liftOpt}_G(f)(g))\big)} \\
\triangleright\text{-notation}: \quad & = p^{:G^A\to H^A} \to \underline{g^{:G^B}} \to g \triangleright \text{liftOpt}_G(f) \triangleright p \triangleright \text{liftOpt}_H(f) \\
\text{omit } (g \to g\,\triangleright): \quad & = p \to \text{liftOpt}_G(f) \,;\, p \,;\, \text{liftOpt}_H(f) \quad .
\end{aligned} \tag{9.49}$$

Note that the last line is similar to Eq. (6.15) but with `liftOpt` instead of `map`:

$$(f^{:A\to B})^{\uparrow F} = p^{:G^A\to H^A} \to f^{\downarrow G} \,;\, p \,;\, f^{\uparrow F} = p \to \text{cmap}_G(f) \,;\, p \,;\, \text{fmap}_F(f) \quad .$$

The laws for filterable contrafunctors are chosen such that $F^A \triangleq G^A \to H^A$ can be shown to obey filtering laws when $H^\bullet$ is a filterable functor and $G^\bullet$ is a filterable contrafunctor.

```
def trace[N: Numeric](matrix: Seq[Seq[N]]): N = ???
```

10.1.3 Pass/fail monads

The type `Option[A]` can be viewed as a collection that can either empty or hold a single value of type `A`. An "iteration" over such a collection will perform a computation at most once:

```
scala> for { x <- Some(123) } yield x * 2     // The computation is performed once.
res0: Option[Int] = Some(246)
```

When an `Option` value is empty, the computation is not performed at all.

```
scala> for { x <- None: Option[Int] } yield x * 2     // The computation is not performed at all.
res1: Option[Int] = None
```

What would a *nested* "iteration" over several `Option` values do? When all of the `Option` values are non-empty, the "iteration" will perform some computations using the wrapped values. However, if even one of the `Option` values happens to be empty, the computed result will be an empty value:

```
scala> for {
         x <- Some(123)
         y <- None
         z <- Some(-1)
       } yield x + y + z
res2: Option[String] = None
```

Computations with `Either` and `Try` values follow the same logic: nested "iteration" will perform no computations unless all values are non-empty. This logic is useful for implementing a series of computations that could produce failures, where any failure should stop all further processing. For this reason (and since they all support the `pure` method and are lawful monads, as this chapter will show), we call the type constructors `Option`, `Either`, and `Try` the **pass/fail monads**.

The following schematic example illustrates this logic:

```
val result: Try[A] = for { // Computations in the ‘Try‘ monad.
  x <- Try(k())            // First computation ‘k()‘, may fail.
  y = f(x)                 // No possibility of failure in this line.
  if p(y)                  // The entire expression will fail if ‘p(y) == false‘.
  z <- Try(g(x, y))        // The computation may also fail here.
  r <- Try(h(x, y, z))     // May fail here as well.
} yield r                  // If ‘r‘ has type ‘A‘ then ‘result‘ has type ‘Try[A]‘.
```

The function `Try()` catches exceptions thrown by its argument. If one of `k()`, `g(x, y)`, or `h(x, y, z)` throws an exception, the corresponding `Try(...)` value will evaluate to a `Failure(...)` case class, and further computations will not be performed. The value `result` will indicate the *first* encountered failure. Only if all `Try(...)` values evaluate to a `Success(...)` case class, the entire expression evaluates to a `result` of type `Success` that wraps a value of type `A`.

Whenever this pattern of computation is found, a functor block gives concise and readable code that replaces a series of nested `if`/`else` or `match`/`case` expressions. A typical situation was shown in Example 3.2.2.4 (Chapter 3), where a "safe integer" computation continues only as long as every result is a success; the chain of operations stops at the first failure. The code of Example 3.2.2.4 introduced custom data type with hand-coded methods such as `add`, `mul`, and `div`. We can now implement equivalent functionality using functor blocks and a standard type `Either[String, Int]`:

```
type Result = Either[String, Int]
def div(x: Int, y: Int): Result = if (y == 0) Left(s"error: $x / $y") else Right(x / y)
def sqrt(x: Int): Result = if (x < 0) Left(s"error: sqrt($x)") else Right(math.sqrt(x).toInt)
val previous: Result = Right(20)  // Start with some given ‘previous‘ value of type ‘Result‘.

scala> val result: Result = for {  // Safe computation: ‘sqrt(1000 / previous - 100) + 20‘.
  x <- previous
  y <- div(1000, x)
```

$$\text{right-hand side :}\quad \text{ftn}^{:Z+Z+(Z+A)\to Z+(Z+A)}\,\text{;}\,\text{ftn} = \,\begin{array}{|c||cc|} & Z & \underline{Z+A} \\ \hline Z & \text{id} & \mathbb{0} \\ Z & \text{id} & \mathbb{0} \\ \underline{Z+A} & \mathbb{0} & \underline{\text{id}} \end{array}\,\text{;}\,\begin{array}{|c||cc|} & Z & A \\ \hline Z & \text{id} & \mathbb{0} \\ Z & \text{id} & \mathbb{0} \\ A & \mathbb{0} & \text{id} \end{array}$$

$$\text{expand id}^{Z+A}\text{ :}\quad = \,\begin{array}{|c||ccc|} & Z & Z & A \\ \hline Z & \text{id} & \mathbb{0} & \mathbb{0} \\ Z & \text{id} & \mathbb{0} & \mathbb{0} \\ Z & \mathbb{0} & \text{id} & \mathbb{0} \\ A & \mathbb{0} & \mathbb{0} & \text{id} \end{array}\,\text{;}\,\begin{array}{|c||cc|} & Z & A \\ \hline Z & \text{id} & \mathbb{0} \\ Z & \text{id} & \mathbb{0} \\ A & \mathbb{0} & \text{id} \end{array} = \,\begin{array}{|c||cc|} & Z & A \\ \hline Z & \text{id} & \mathbb{0} \\ Z & \text{id} & \mathbb{0} \\ Z & \text{id} & \mathbb{0} \\ A & \mathbb{0} & \text{id} \end{array}\quad.$$

The two sides of the associativity law are equal.

When it works, the technique of Curry-Howard code inference gives much shorter proofs than explicit derivations:

Example 10.2.3.2 Verify that the `Reader` monad, $F^A \triangleq Z \to A$, satisfies the associativity law.

Solution The type signature of `flatten` is $(Z \to Z \to A) \to Z \to A$. Both sides of the law (10.6) are functions with the type signature $(Z \to Z \to Z \to A) \to Z \to A$. By code inference with typed holes, we find that there is only one fully parametric implementation of this type signature, namely

$$p^{:Z\to Z\to Z\to A} \to z^{:Z} \to p(z)(z)(z)\quad.$$

So, both sides of the law must have the same code, and the law holds.

Example 10.2.3.3 Show that the `List` monad ($F^A \triangleq \text{List}^A$) satisfies the associativity law.

Solution The `flatten[A]` method has the type signature $\text{ftn}^A : \text{List}^{\text{List}^A} \to \text{List}^A$ and concatenates the nested lists in their order. Let us first show a more visually clear (but less formal) proof of the associativity law. Both sides of the law are functions of type $\text{List}^{\text{List}^{\text{List}^A}} \to \text{List}^A$. We can visualize how both sides of the law are applied to a triple-nested list value p defined by

$$p \triangleq [[[x_{11}, x_{12}, ...], [x_{21}, x_{22}, ...], ...], [[y_{11}, y_{12}, ...], [y_{21}, y_{22}, ...], ...], ...]\quad,$$

where all x_{ij}, y_{ij}, ... have type A. Applying $\text{ftn}^{\uparrow\text{List}}$ flattens the inner lists and produces

$$p \triangleright \text{ftn}^{\uparrow\text{List}} = [[x_{11}, x_{12}, ..., x_{21}, x_{22}, ...], [y_{11}, y_{12}, ..., y_{21}, y_{22}, ...], ...]\quad.$$

Flattening that result gives a list of all values x_{ij}, y_{ij}, ..., in the order they appear in p:

$$p \triangleright \text{ftn}^{\uparrow\text{List}} \triangleright \text{ftn} = [x_{11}, x_{12}, ..., x_{21}, x_{22}, ..., y_{11}, y_{12}, ..., y_{21}, y_{22}, ..., ...]\quad.$$

Applying $\text{ftn}^{\text{List}^A}$ to p will flatten the outer lists,

$$p \triangleright \text{ftn}^{\text{List}^A} = [[x_{11}, x_{12}, ...], [x_{21}, x_{22}, ...], ..., [y_{11}, y_{12}, ...], [y_{21}, y_{22}, ...], ...]\quad.$$

Flattening that value results in $p \triangleright \text{ftn}^{\text{List}^A} \triangleright \text{ftn} = [x_{11}, x_{12}, ..., x_{21}, x_{22}, ..., y_{11}, y_{12}, ..., y_{21}, y_{22}, ..., ...]$. This is exactly the same as $p \triangleright \text{ftn}^{\uparrow\text{List}} \triangleright \text{ftn}$: namely, the list of all values in the order they appear in p.

A formal proof of the associativity law is by an explicit derivation. Using the recursive type definition $\text{List}^A \triangleq \mathbb{1} + A \times \text{List}^A$, we can define `flatten` as a recursive function:

$$\text{ftn}^A \triangleq \,\begin{array}{|c||c|} & \mathbb{1} + \text{List}^A \times \text{List}^{\text{List}^A} \\ \hline \mathbb{1} & 1 \to 1 + \mathbb{0} \\ \text{List}^A \times \text{List}^{\text{List}^A} & h^{:\text{List}^A} \times t^{:\text{List}^{\text{List}^A}} \to h \mathbin{+\!\!+} t \triangleright \overline{\text{ftn}} \end{array}\quad,$$

Although the `pure` method can be replaced by a simpler "wrapped unit" value (wu_M), having no laws, derivations turn out to be easier when using pu_M.

The `Pointed` typeclass requires the `pure` method to satisfy the naturality law (8.8). A full monad's `pure` method must satisfy that law, in addition to the identity laws.

Just as some useful semigroups are not monoids, there exist some useful semimonads that are not full monads. A simple example is the `Writer` semimonad $F^A \triangleq A \times W$ whose type W is a semigroup but not a monoid (see Exercise 10.2.9.1).

10.2.5 The monad identity laws in terms of `pure` and `flatten`

Since the laws of semimonads are simpler when formulated via the `flatten` method, let us convert the identity laws to that form. We use the code for `flatMap` in terms of `flatten`,

$$\text{flm}_M(f^{:A\to M^B}) = f^{\uparrow M} \fatsemi \text{ftn}_M \quad .$$

Begin with the left identity law of `flatMap`, written as

$$\text{pu}_M \fatsemi \text{flm}_M(f) = f \quad .$$

$$\begin{tikzcd} & M^{M^A} \arrow[rd, "\text{ftn}^A"] & \\ M^A \arrow[ru, "\text{pu}^{M^A}"] \arrow[rr, "\text{id}"'] & & M^A \end{tikzcd}$$

Since this law holds for arbitrary f, we can set $f \triangleq \text{id}$ and get

$$\text{pu}_M \fatsemi \text{ftn}_M = \text{id}^{:M^A\to M^A} \quad . \tag{10.9}$$

This is the **left identity law** of `flatten`. Conversely, if Eq. (10.9) holds, we can compose both sides with an arbitrary function $f^{:A\to M^B}$ and recover the left identity law of `flatMap` (Exercise 10.2.9.2).

The **right identity law** of `flatten` is written as

$$\begin{tikzcd} & M^{M^A} \arrow[rd, "\text{ftn}^A"] & \\ M^A \arrow[ru, "(\text{pu}^A)^{\uparrow M}"] \arrow[rr, "\text{id}"'] & & M^A \end{tikzcd}$$

$$\text{flm}_M(\text{pu}_M) = \text{pu}_M^{\uparrow M} \fatsemi \text{ftn}_M \overset{!}{=} \text{id} \quad . \tag{10.10}$$

In the next section, we will see a reason why these laws have their names.

10.2.6 Monad laws in terms of Kleisli functions

A **Kleisli function** is a function with type signature $A \to M^B$ where M is a monad. We first encountered Kleisli functions in Section 9.2.3 when deriving the laws of filterable functors using the `liftOpt` method. At that point, M was the simple `Option` monad. We found that functions of type $A \to \mathbb{1} + B$ can be composed using the Kleisli composition denoted by $\diamond_{\text{Opt}}$ (see page 309). Later, Section 9.4.2 stated the general properties of Kleisli composition. We will now show that the Kleisli composition gives a useful way of formulating the laws of a monad.

The Kleisli composition operation for a monad M, denoted $\diamond_M$, is a function with type signature

$$\diamond_M : (A \to M^B) \to (B \to M^C) \to A \to M^C \quad .$$

This resembles the forward composition of ordinary functions, $(\fatsemi) : (A \to B) \to (B \to C) \to A \to C$, except for different types of functions. If M is a monad, the implementation of $\diamond_M$ is

```
def <>[M[_]: Monad, A,B,C](f: A => M[B], g: B => M[C]): A => M[C] =
  { x => f(x).flatMap(g) }
```

$$f \diamond_M g \triangleq f \fatsemi \text{flm}_M(g) \quad . \tag{10.11}$$

The Kleisli composition can be equivalently expressed by a functor block code as

```
def <>[M[_]: Monad, A,B,C](f: A => M[B], g: B => M[C]): A => M[C] = { x =>
  for {
    y <- f(x)
    z <- g(y)
  } yield z
}
```

This example shows that Kleisli composition is a basic part of functor block code: it expresses the chaining of two consecutive "source" lines.

Let us now derive the laws of Kleisli composition $\diamond_M$, assuming that the monad laws hold for M.

Statement 10.2.6.1 For a lawful monad M, the Kleisli composition $\diamond_M$ satisfies the identity laws

$$\text{left identity law of } \diamond_M : \quad \mathrm{pu}_M \diamond_M f = f \quad , \quad \forall f^{:A\to M^B} \quad , \tag{10.12}$$

$$\text{right identity law of } \diamond_M : \quad f \diamond_M \mathrm{pu}_M = f \quad , \quad \forall f^{:A\to M^B} \quad . \tag{10.13}$$

Proof We may assume that Eqs. (10.7)–(10.8) hold. Using the definition (10.11), we find

$$\begin{aligned} \text{left identity law of } \diamond_M\text{, should equal } f : \quad & \mathrm{pu}_M \diamond_M f = \underline{\mathrm{pu}_M \,⨾\, \mathrm{flm}_M(f)} \\ \text{use Eq. (10.7)} : \quad & = f \quad , \\ \text{right identity law of } \diamond_M\text{, should equal } f : \quad & f \diamond_M \mathrm{pu}_M = f \,⨾\, \underline{\mathrm{flm}_M(\mathrm{pu}_M)} \\ \text{use Eq. (10.8)} : \quad & = f \,⨾\, \mathrm{id} = f \quad . \end{aligned}$$

The following statement and the identity law (10.8) show that `flatMap` can be viewed as a "lifting",

$$\mathrm{flm}_M : (A \to M^B) \to (M^A \to M^B) \quad ,$$

from Kleisli functions $A \to M^B$ to M-lifted functions $M^A \to M^B$, except that Kleisli functions must be composed using $\diamond_M$, while pu_M plays the role of the Kleisli-identity function.

Statement 10.2.6.2 For a lawful monad M, the `flatMap` method satisfies the composition law

$$\begin{tikzcd} & M^B \arrow[dr, "\mathrm{flm}_M(g)"] & \\ M^A \arrow[ur, "\mathrm{flm}_M(f)"] \arrow[rr, "\mathrm{flm}_M(f \diamond_M g)"'] & & M^C \end{tikzcd}$$

$$\mathrm{flm}_M(f \diamond_M g) = \mathrm{flm}_M(f) \,⨾\, \mathrm{flm}_M(g) \quad .$$

Proof We may use Eq. (10.4) since M is a lawful monad. A direct calculation yields the law:

$$\begin{aligned} \text{expect to equal } \mathrm{flm}_M(f) \,⨾\, \mathrm{flm}_M(g) : \quad & \mathrm{flm}_M(f \diamond_M g) = \mathrm{flm}_M(f \,⨾\, \mathrm{flm}_M(g)) \\ \text{use Eq. (10.4)} : \quad & = \mathrm{flm}_M(f) \,⨾\, \mathrm{flm}_M(g) \quad . \end{aligned}$$

The following statement motivates calling Eq. (10.4) an "associativity" law.

Statement 10.2.6.3 For a lawful monad M, the Kleisli composition $\diamond_M$ satisfies the **associativity law**

$$(f \diamond_M g) \diamond_M h = f \diamond_M (g \diamond_M h) \quad , \quad \forall f^{:A\to M^B}, g^{:B\to M^C}, h^{:C\to M^D} \quad . \tag{10.14}$$

So, we may write $f \diamond_M g \diamond_M h$ unambiguously with no parentheses.

Proof Substitute Eq. (10.11) into both sides of the law:

$$\begin{aligned} \text{left-hand side} : \quad & (\underline{f \diamond_M g}) \diamond_M h = (f \,⨾\, \mathrm{flm}_M(g)) \underline{\diamond_M h} = f \,⨾\, \mathrm{flm}_M(g) \,⨾\, \mathrm{flm}_M(h) \quad , \\ \text{right-hand side} : \quad & \underline{f \diamond_M} (g \diamond_M h) = f \,⨾\, \underline{\mathrm{flm}_M(g \diamond_M h)} \\ \text{use Statement 10.2.6.2} : \quad & = f \,⨾\, \mathrm{flm}_M(g) \,⨾\, \mathrm{flm}_M(h) \quad . \end{aligned}$$

Both sides of the law are now equal.

We find that the properties of the operation $\diamond_M$ are similar to the identity and associativity properties of the function composition $f \,⨾\, g$ except for using pu_M instead of the identity function.[8]

Since the Kleisli composition describes the chaining of consecutive lines in functor blocks, its associativity means that multiple lines are chained unambiguously. For example, this code:

[8]It means that Kleisli functions satisfy the properties of morphisms of a category; see Section 9.4.3.

```
  def up: ReaderT[M, R, A] = ReaderT(r => Monad[M].pure(t(r)))
}
```

The lifts are written in the code notation as

$$\text{foreign lift :}\quad \text{flift} : M^A \to T^{M,A}_{\text{Reader}}\quad,\qquad \text{flift}\,(m^{:M^A}) \triangleq \_^{:R} \to m = \text{pu}_{\text{Reader}}(m)\quad,$$
$$\text{base lift :}\quad \text{blift} : (R \to A) \to T^{M,A}_{\text{Reader}}\quad,\qquad \text{blift}\,(t^{:R\to A}) \triangleq r^{:R} \to \text{pu}_M(t(r)) = t\,⨾\,\text{pu}_M\quad.$$

We have seen in Section 10.1.5 that getting data out of the `Reader` monad requires a runner θ_{Reader}, which is a function that calls `run` on some value of type R (i.e., injects `Reader`'s dependency value). In later sections 10.1.7 and 10.1.9, we have seen other monads that need runners. Generally, a runner for a monad M is a function of type $M^A \to A$. So, we may expect that the foreign monad M could have its own runner θ_M. How can we combine M's runner (θ_M) with `Reader`'s runner? Since the type of T^M_{Reader} is a functor composition of `Reader` and M, the runners can be used independently of each other. We can first run the effect of M and then run the effect of the `Reader`:

$$(\theta_M^{\uparrow\text{Reader}}\,⨾\,\theta_{\text{Reader}}) : (R \to M^A) \to A\quad.$$

Alternatively, we can run `Reader` first (injecting the dependency) and then run M's effect:

$$(\theta_{\text{Reader}}\,⨾\,\theta_M) : (R \to M^A) \to A\quad.$$

These runners commute because of the naturality law of θ_{Reader}, which holds for any $f^{:A\to B}$:

$$f^{\uparrow\text{Reader}}\,⨾\,\theta_{\text{Reader}} = \theta_{\text{Reader}}\,⨾\,f\quad,\qquad\text{so}\qquad \theta_M^{\uparrow\text{Reader}}\,⨾\,\theta_{\text{Reader}} = \theta_{\text{Reader}}\,⨾\,\theta_M\quad.$$

The `EitherT` transformer is similar to `TryT` since the type `Try[A]` is equivalent to `Either[Throwable, A]`.

The `WriterT` transformer It is easier to begin with the `flatten` method for the transformed monad $T^A \triangleq M^{A\times W}$, which has type signature

$$\text{ftn}_T : M^{M^{A\times W}\times W} \to M^{A\times W}\quad,\qquad \text{ftn}_T\big(t^{:M^{M^{A\times W}\times W}}\big) = ???^{:M^{A\times W}}\quad.$$

Since M is an unknown, arbitrary monad, the only way of computing a value of type $M^{A\times W}$ is by using the given value t. The only way to get a value of type A wrapped in $M^{A\times W}$ is by extracting the type A from inside $M^{M^{A\dots}\dots}$. So, we need to flatten the two layers of M that are present in the type of t. However, we cannot immediately apply M's `flatten` method to t because t's type is not of the form M^{M^X} with some X. To bring it to that form, we use M's `map` method:

$$t \triangleright (m^{:M^{A\times W}} \times w^{:W} \to m \triangleright (p^{:A\times W} \to p \times w)^{\uparrow M})^{\uparrow M} : M^{M^{A\times W\times W}}\quad.$$

Now the type is well adapted to using both M's and `Writer`'s flatten methods:

$$\text{ftn}_T\big(t^{:M^{M^{A\times W}\times W}}\big) = t \triangleright (m^{:M^{A\times W}} \times w^{:W} \to m \triangleright (p^{:A\times W} \to p\times w)^{\uparrow M})^{\uparrow M} \triangleright \text{ftn}_M \triangleright (\text{ftn}_{\text{Writer}})^{\uparrow M}\quad,$$
$$\begin{aligned}\text{ftn}_T &= (m^{:M^{A\times W}} \times w^{:W} \to m \triangleright (p^{:A\times W} \to p\times w)^{\uparrow M})^{\uparrow M}\,⨾\,\text{ftn}_M\,⨾\,(\text{ftn}_{\text{Writer}})^{\uparrow M}\\ &= \text{flm}_M(m\times w \to m \triangleright (p \to p\times w)^{\uparrow M}\,⨾\,\text{ftn}^{\uparrow M}_{\text{Writer}})\\ &= \text{flm}_M(m\times w \to m \triangleright (a\times w_2 \to a\times(w\oplus w_2)))\quad.\end{aligned}$$

Translating this formula to Scala, we obtain the code of `flatMap`:

```
final case class WriterT[M[_]: Monad : Functor, W: Monoid, A](t: M[(A, W)]) {
  def map[B](f: A => B): WriterT[M, W, B] = WriterT(t.map { case (a, w) => (f(a), w) })
  def flatMap[B](f: A => WriterT[M, W, B]): WriterT[M, W, B] = WriterT(
    t.flatMap { case (a, w) => f(a).t.map { case (b, w2) => (b, w |+| w2) }
  })
}
```