# Tracing monadic computations and representing effects

Maciej Piróg and Jeremy Gibbons
Department of Computer Science
University of Oxford
Maciej.Pirog@cs.ox.ac.uk Jeremy.Gibbons@cs.ox.ac.uk

In functional programming, monads are supposed to encapsulate computations, effectfully producing the final result, but keeping to themselves the means of acquiring it. For various reasons, we sometimes want to reveal the internals of a computation. To make that possible, in this paper we introduce monad transformers that add the ability to automatically accumulate observations about the course of execution as an effect. We discover that if we treat the resulting trace as the actual result of the computation, we can find new functionality in existing monads, notably when working with non-terminating computations.

## 1 Introduction

Consider a simple database in which different users store data under keys. This can be represented in Haskell as a data structure of type $[(User, [(Key, Data)])]$. We can define a function *getData* which looks up a value for a specified user and a key.

$$getData :: [(User, [(Key, Data)])] \to User \to Key \to Maybe\ Data$$
$$getData\ db\ u\ k = lookup\ u\ db \gg\!= lookup\ k$$

If a user $u$ has an entry $d$ associated with a key $k$ in the database, *getData u k* returns *Just d*; otherwise, it returns *Nothing*. In the latter case we might want to inform the user 'why' the program is not able to deliver data: they might have misspelled their username, which means that *lookup u* will fail, or they might have tried to read from a missing key, which means that *lookup k* will fail. Unfortunately, the *Maybe* monad does not allow one to observe at what point a failing program actually fails. We need to structure our function using a more sophisticated monad.

What are the desired properties of such a monad? For sure, we want it to employ the same kind of effects as *Maybe*, so that we do not have to alter the logic of our program. We would like to have a *lift* operation, which allows us to automatically translate some operations from *Maybe* into the new monad, so that it is not necessary to rewrite standard functions like *lookup*. But most importantly, it must also reveal some observations about the course of execution (a *trace*), such as the number of successful subcomputations, from which we can extract the desired information about the point of failure. One possibility is to explicitly accumulate such observations inside the computation, using monads like *Writer* or *Exception*. In this article, however, we take a different approach: we automate the accumulation inside the monadic structure. The accumulation is transparent inside the computation; that is, it cannot affect the course of execution, and is revealed only on the outside.

We aim for maximum genericity: we construct transformers that add traces to arbitrary monads. Our main tools are free monads (Section 2), which have the ability to represent the very structure of monadic computations. They provide, in a sense, the most informative traces possible. Then, we introduce a transformer, called *Nest*, which allows one to mix the free and effectful computations provided by a monad (Section 3). The genericity pays off, and we find a number of applications for tracing monads:

J. Chapman and P. B. Levy (Eds.): Fourth Workshop on
Mathematically Structured Functional Programming (MSFP 2012).
EPTCS 76, 2012, pp. 90–111, doi:10.4204/EPTCS.76.8

- Traces of computations can be interpreted in different ways. In Section 4.3 we show an example in which, by revealing the inner structure of a computation in the *List* monad, we can define the Prolog *cut* operator.
- Computations encapsulated in monadic expressions are often monolithic. They are supposed to produce final values only, so there is little space for non-termination. But, assuming non-strict semantics, we can see computations as entities that lazily unfold a trace. So, even if a computation is non-terminating, we can benefit from its infinite trace. If we interpret free parts of a *Nest* value as terms, we discover that *Nest* is a generalisation of Capretta's partiality monad (see Section 4.1).
- For the same reasons, traces are a useful tool for modelling impure interaction with the environment. In Section 6, we sketch out some future work on a novel, coinductive approach to the functional semantics of effects.

## 2 Simple tracing with free monads

Moggi [19] called monads *notions of computation*, because they describe computational effects in a way that abstracts from the type of values produced by a computation. In a sense, from the categorical point of view, monads abstract even from the exact values produced by the computation, since *return* and *join* are natural transformations. It is the *bind* operator (defined as $m \gg\!= f = \mathit{join}\ (\mathit{fmap}\ f\ m)$) that mixes the functorial (looking only at the values) *fmap* and parametric (looking only at the structure) *join*. The laws for monads and functors entail the following equality, called *naturality* of join.

$$\mathit{fmap}\ f \circ \mathit{join} \equiv \mathit{join} \circ \mathit{fmap}\ (\mathit{fmap}\ f)$$

Read as a transformation from left to right, it allows one to move occurrences of *join* to the left of a composition, and occurrences of *fmap* to the right. This way, we can split computations with multiple steps into a "mapping" part and a "joining" part. For example, consider a computation $m \gg\!= f \gg\!= g$. It can be split as follows.

$$m \gg\!= f \gg\!= g \equiv (\mathit{join} \circ \mathit{fmap}\ g \circ \mathit{join} \circ \mathit{fmap}\ f)\ m \equiv (\mathit{join} \circ \mathit{join} \circ \mathit{fmap}\ (\mathit{fmap}\ g) \circ \mathit{fmap}\ f)\ m$$

Intuitively, we can see the "mapping" part (that is, $\mathit{fmap}\ (\mathit{fmap}\ g) \circ \mathit{fmap}\ f$) as the construction of the computation and the "joining" part ($\mathit{join} \circ \mathit{join}$) as the execution of effects.

Our first approach to tracing is to capture the mapping part of a computation. We suspend execution of the *join* operators of the monad, so that we can isolate and examine the elements from which the computation is composed.

### 2.1 Free monads

The type of the mapping part is different for different numbers of nested occurrences of the *fmap* function, and so for computations consisting of different numbers of steps. For example,

$$\begin{array}{ll}
\mathit{Just}\ \texttt{'a'} & :: \mathit{Maybe}\ \mathit{Char} \\
\mathit{fmap}\ \mathit{Just}\ (\mathit{Just}\ \texttt{'a'}) & :: \mathit{Maybe}\ (\mathit{Maybe}\ \mathit{Char}) \\
\mathit{fmap}\ (\mathit{fmap}\ \mathit{Just})\ (\mathit{fmap}\ \mathit{Just}\ (\mathit{Just}\ \texttt{'a'})) & :: \mathit{Maybe}\ (\mathit{Maybe}\ (\mathit{Maybe}\ \mathit{Char}))
\end{array}$$

A reasonable class of datatypes in which to store such expressions are the *free monads*, also known as $f$*-generated trees*. Allowing ourselves to define monads in terms of *fmap*, *return*, and *join*, rather than the *return* and $\gg\!=$ required by Haskell, we can write:

$$
\begin{array}{l}
\textbf{data}\ \mathit{Free}\ f\ a = \mathit{Wrap}\ (f\ (\mathit{Free}\ f\ a)) \mid \mathit{Return}\ a \\
\textbf{instance}\ \mathit{Functor}\ f \Rightarrow \mathit{Functor}\ (\mathit{Free}\ f)\ \textbf{where} \\
\quad \mathit{fmap}\ g\ (\mathit{Return}\ a) = \mathit{Return}\ (g\ a) \\
\quad \mathit{fmap}\ g\ (\mathit{Wrap}\ f) \quad = \mathit{Wrap}\ (\mathit{fmap}\ (\mathit{fmap}\ g)\ f) \\
\textbf{instance}\ \mathit{Functor}\ f \Rightarrow \mathit{Monad}\ (\mathit{Free}\ f)\ \textbf{where} \\
\quad \mathit{return} \qquad\quad\ \ = \mathit{Return} \\
\quad \mathit{join}\ (\mathit{Return}\ f) \quad = f \\
\quad \mathit{join}\ (\mathit{Wrap}\ f) \qquad = \mathit{Wrap}\ (\mathit{fmap}\ \mathit{join}\ f)
\end{array}
$$

Each level of the type constructor of a monad corresponds to a level in the *Free* data structure. We store *Just* `'a'` :: *Maybe Char* as *Wrap* (*Just* (*Return* `'a'`)), and *Just* (*Just* (`'a'`)) :: *Maybe* (*Maybe Char*) as *Wrap* (*Just* (*Wrap* (*Just* (*Return* `'a'`)))), and so on.

## 2.2 Monad transformers with *drop*

The type *Free m a* can be seen as a datatype of terms generated by the signature *m* (a functor) and a set of variables *a*. Even if *m* is a monad, *Free m* cannot depend on any effects provided by *m*; the join operation for *Free m* performs only substitution, and is independent of the join for *m*.

In order to couple a monad *m* and the *m*-generated free monad, we need the notion of *monad transformers* [16]. A monad transformer with *drop* is a relation between two monads, *m* and *t*, characterised by two functions, *lift* and *drop*, which translate computations in *m* into computations in *t*, and *vice versa*. The relationship can be defined as the following two-parameter Haskell type class. Though in functional programming *drop* is rarely considered to be a member of *MonadTrans*, it plays an important role here. For the moment, we forget that we usually insist that monad transformers are subject to a certain set of algebraic laws.

$$
\begin{array}{l}
\textbf{class}\ (\mathit{Monad}\ m, \mathit{Monad}\ t) \Rightarrow \mathit{MonadTrans}\ m\ t \mid t \to m\ \textbf{where} \\
\quad \mathit{lift}\ \ :: m\ a \to t\ a \\
\quad \mathit{drop} :: t\ a\ \ \to m\ a
\end{array}
$$

For any monad *m*, we define an instance of *MonadTrans m* (*Free m*) as follows. The *lift* operation is straightforward, as it only wraps the value and maps the *Return* constructor. The *drop* operation traverses the structure and flattens each level, thus performing suspended *bind*s of *m*.

$$
\begin{array}{l}
\textbf{instance}\ (\mathit{Functor}\ m, \mathit{Monad}\ m) \Rightarrow \mathit{MonadTrans}\ m\ (\mathit{Free}\ m)\ \textbf{where} \\
\quad \mathit{lift} \qquad\qquad\quad = \mathit{Wrap} \circ \mathit{fmap}\ \mathit{Return} \\
\quad \mathit{drop}\ (\mathit{Return}\ a) = \mathit{return}\ a \\
\quad \mathit{drop}\ (\mathit{Wrap}\ m) \ \ = m \gg\!= \mathit{drop}
\end{array}
$$

## 2.3 Examples

Now, we can test our tracing free monad transformer on the *Maybe* monad. The *lift* function allows us to translate any computation in *m* into a computation in *Free m*. To get the original, non-traced computation back, we use the *drop* function.

We can see every composition *Wrap* ∘ *Just* as a “tick”, given for each lifted successful subcomputation. The trace forms a unary counter, storing the number of ticks. The final value of the computation is stored in the last *Wrap*. Consider the following conversation with the Haskell interactive shell.

$\triangleright$ **do** $\{$*lift* $(Just\ 2)$; *lift* $(Just\ 4)$; *lift Nothing*$\}$
$Wrap\ (Just\ (Wrap\ (Just\ (Wrap\ Nothing))))$

$\triangleright$ *drop* (**do** $\{$*lift* $(Just\ 2)$; *lift* $(Just\ 4)$; *lift Nothing*$\}$)
$Nothing$

Similarly, for the *Writer* monad, we can get a list of all the values appended to the monoid followed by the final return value. (For brevity, we show $Writer\ (a, Sum\ s)$ as $(s, a)$.)

$\triangleright$ **do** $\{$*lift* $(tell\ (Sum\ 2))$; *lift* $(tell\ (Sum\ 3))$; *lift* $(tell\ (Sum\ 7))$; *return* `'a'`$\}$
$Wrap\ (2, Wrap\ (3, Wrap\ (7, Return$ `'a'`$)))$

$\triangleright$ *drop* (**do** $\{$*lift* $(tell\ (Sum\ 2))$; *lift* $(tell\ (Sum\ 3))$; *lift* $(tell\ (Sum\ 7))$; *return* `'a'`$\}$)
$(12,$ `'a'`$)$

An important thing to notice is that non-terminating computations in *Writer* do not make much sense. For example, the following computation just diverges.

$\triangleright$ **let** $w\ n =$ **do** $\{tell\ (Sum\ n);\ w\ (n+1)\}$ **in** $w\ 0$
$(\ast\ast\ast$ Exception: stack overflow

In contrast, with the tracing version of *Writer*, we can enjoy an infinite stream of actions that happen during the execution:

$\triangleright$ **let** $w\ n =$ **do** $\{lift\ (tell\ (Sum\ n));\ w\ (n+1)\}$ **in** $w\ 0$
$Wrap\ (0, Wrap\ (1, Wrap\ (2, Wrap\ (3, Wrap\ (4, Wrap\ (5, Wrap\ (6, Wrap\ (7, Wrap\ (8, Wrap \ldots$

## 3 More advanced tracing with free structures

Tracing computations with free structures is not very flexible: every bind and every join is suspended, creating a new level of structure every time a monadic action is called. In some circumstances we would like to trace only selected parts of the computation—perhaps we are confident that the other parts always succeed, or we want to treat selected pieces of the computation monolithically, and we are not interested in a fine-grained report about their behaviour.

Another issue is the algebraic properties of monad transformers with *drop*. Intuitively, a pair of monads *m* and *t* are related as a monad transformer if *t* incorporates at least the same effects as *m*. This is usually formalised with the following equalities [12].

$$\begin{aligned}
&lift\ (return\ a) &&= return\ a\\
&lift\ (m \gg\!= f) &&= lift\ m \gg\!= (lift \circ f)\\
&drop\ (return\ a) &&= return\ a\\
&drop\ (lift\ m \gg\!= f) &&= m \gg\!= drop \circ f
\end{aligned}$$

The first two equalities state that *lift* is a monad morphism. The instance $MonadTrans\ m\ (Free\ m)$ from Section 2.2 does not have this property. For example, $lift\ (Just\ 1) \gg lift\ (Just\ 2)$ is equal to $Wrap\ (Just\ (Wrap\ (Just\ (Return\ 2))))$, while $lift\ (Just\ 1 \gg Just\ 2)$ is equal to $Wrap\ (Just\ (Return\ 2))$.

For these two reasons, we abandon the idea of using free monads directly to trace computations. Instead, in this section we introduce a general class, *MonadTrace*, which allows one to specify the points at which to make observations about the execution, and a corresponding version of the *Free* monad that is a proper monad transformer.

### 3.1 The *MonadTrace* class

The *MonadTrace* class introduces a single monadic value, *mark*. Intuitively, this is an operation that stores the current state of execution in the trace.

$$\begin{array}{l}\textbf{class}\ \mathit{Monad}\ t \Rightarrow \mathit{MonadTrace}\ t\ \textbf{where}\\ \quad \mathit{mark} :: t\ ()\end{array}$$

We call a monad $v$ a *tracer* of a monad $m$, if $m$ and $v$ form a monad transformer ($\mathit{MonadTrans}\ m\ v$), and $v$ is an instance of *MonadTrace*. Additionally, *lift*, *drop* and *mark* should satisfy the following equalities.

$$\begin{array}{lll}\mathit{lift} & (\mathit{return}\ a) & = \mathit{return}\ a\\ \mathit{lift} & (m \gg\!= f) & = \mathit{lift}\ m \gg\!= (\mathit{lift} \circ f)\\ \mathit{drop} & (\mathit{return}\ a) & = \mathit{return}\ a\\ \mathit{drop} & (v \gg\!= g) & = \mathit{drop}\ v \gg\!= (\mathit{drop} \circ g)\\ \mathit{drop}\ \mathit{mark} & & = \mathit{return}\ ()\end{array}$$

The tracer $v$ cannot perform more effects than the monad $m$, except for tracing with the *mark* operation. Nonetheless, tracing does not affect the course of computation in any way observable from inside of the $v$-computation, hence both *lift* and *drop* are monad morphisms. Note that the laws entail $\mathit{drop} \circ \mathit{lift} = \mathit{id}$.

We use the *mark* gadget wherever we want to make an observation. In the following example, the intuitive semantics of a tracer for the *Maybe* monad is that we get a tick whenever the computation is still successful when placing a mark.

$$\textbf{do}\ \{x \leftarrow \mathit{lift}\ m_0;\ y \leftarrow \mathit{lift}\ m_1;\ \mathit{mark};\ z \leftarrow \mathit{lift}\ m_2;\ \mathit{mark};\ \mathit{return}\ (x+y+z)\}$$

That means that if $m_0$ is successful, but $m_1$ fails, no ticks are stored in the trace (intuitively, it is equivalent to *Nothing*). Only if both $m_0$ and $m_1$ are successful, the *mark* operation stores this success (the trace is of the form $\mathit{Just}\ a$, where $a$ is a result of the rest of the computation). If $m_2$ is also successful, the second call to *mark* stores this information in the trace (and the trace is of the form $\mathit{Just}\ (\mathit{Just}\ a)$, where $a = \mathit{return}\ (x+y+z) = \mathit{Just}\ (x+y+z)$).

We also define a convenient function, *mind*, to perform a traced lifting.

$$\begin{array}{l}\mathit{mind} :: (\mathit{MonadTrans}\ m\ v,\ \mathit{MonadTrace}\ v) \Rightarrow m\ a \to v\ a\\ \mathit{mind}\ m = \textbf{do}\ \{x \leftarrow \mathit{lift}\ m;\ \mathit{mark};\ \mathit{return}\ x\}\end{array}$$

Then the previous example can be written more concisely:

$$\textbf{do}\ \{x \leftarrow \mathit{lift}\ m_0;\ y \leftarrow \mathit{mind}\ m_1;\ z \leftarrow \mathit{mind}\ m_2;\ \mathit{return}\ (x+y+z)\}$$

### 3.2 The *Nest* monad

Free monads allow one to capture the structure of an $m$-computation as data; but for tracing, we need also to be able to perform some parts of the computations (the lifted ones) immediately. This suggests considering the composition of the two monads $m$ and $\mathit{Free}\ m$, in one order or the other. In fact, because we want the lifted parts to be performed immediately, the appropriate order of composition is to have $m$ on the outside and $\mathit{Free}\ m$ inside. We therefore define:

$$\begin{array}{l}
\textbf{newtype}\ \mathit{Nest}\ m\ a = \mathit{Nest}\{\mathit{unNest} :: m\ (\mathit{Free}\ m\ a)\} \\
\textbf{instance}\ \mathit{Functor}\ m \Rightarrow \mathit{Functor}\ (\mathit{Nest}\ m)\ \textbf{where} \\
\quad \mathit{fmap}\ f\ (\mathit{Nest}\ m) = \mathit{Nest}\ (\mathit{fmap}\ (\mathit{fmap}\ f)\ m)
\end{array}$$

It remains to show that *Nest* can be given the structure of a monad. We do this using Jones and Duponcheel's *prod* construction [14]: given monad $M$ with unit $return_M$ and multiplication $join_M$, and similarly monad $F$ with $return_F$ and $join_F$, the composition $M\ F$ forms a monad with unit and multiplication given by

$$\begin{array}{ll}
return & = return_M \circ return_F \\
join & = join_M \circ fmap_M\ prod
\end{array}$$

Diagrammatically:

$$\begin{tikzcd}
1 \arrow[r, "return_F"] \arrow[dr, "return"'] & F \arrow[d, "return_M\ F"] \\
& MF
\end{tikzcd}
\qquad
\begin{tikzcd}
MFMF \arrow[r, "M\ prod"] \arrow[dr, "join"'] & MMF \arrow[d, "join_M\ F"] \\
& MF
\end{tikzcd}$$

provided that natural transformation $prod :: F\ M\ F \dot{\to} M\ F$ satisfies the three properties

$$\begin{array}{lll}
prod \circ return_F & = id & (1) \\
prod \circ fmap_F\ return & = return_M & (2) \\
prod \circ fmap_F\ join & = join \circ prod & (3)
\end{array}$$

Diagrammatically:

$$\begin{tikzcd}
MF \arrow[r, "return_F\ MF"] \arrow[dr, equal] & FMF \arrow[d, "prod"] \\
& MF
\end{tikzcd}
\qquad
\begin{tikzcd}
F \arrow[r, "F\ return"] \arrow[dr, "return_M\ F"'] & FMF \arrow[d, "prod"] \\
& MF
\end{tikzcd}
\qquad
\begin{tikzcd}
FMFMF \arrow[r, "prod\ MF"] \arrow[d, "F\ join"] & MFMF \arrow[d, "join"] \\
FMF \arrow[r, "prod"] & MF
\end{tikzcd}$$

(In fact, the multiplication $join_F$ of $F$ is not used; all that is required of $F$ is for it to be a *premonad*.) It turns out that the definition of *prod* is—if not quite forced then at least—very strongly suggested by the necessity of satisfying these three properties.

In our case, $M$ is the monad $m$, and $F$ is the datatype *Free m* from Section 2. For brevity, let us write $M$ for $fmap_M$, and similarly $F$ for $fmap_F$; let us also write the coproduct type former as $+$ and the coproduct morphism as $\nabla$, so that we can express the two constructors as one composite,

$$\mathit{Wrap} \nabla \mathit{Return} :: M\ F + 1 \dot{\to} F$$

and name its inverse,

$$out_F :: F \dot{\to} M\ F + 1$$

Recall that the unit of the monad *Free m* is the constructor *Return*:

$$return_F = \mathit{Return}$$

Without loss of generality, we let

$$prod \circ (Wrap \triangledown Return) = prod_1 \triangledown prod_2$$

so that

$$\begin{aligned} prod_1 &= prod \circ Wrap \\ prod_2 &= prod \circ Return \end{aligned}$$

The definition of $prod_2$ is forced:

$$\begin{aligned} & prod_2 \\ = \quad & \{ \text{definition of } prod_2 \} \\ & prod \circ Return \\ = \quad & \{ F \text{ as a premonad: } return_F = Return \} \\ & prod \circ return_F \\ = \quad & \{ \text{property (1)} \} \\ & id \end{aligned}$$

Now consider property (2). Expanding the left-hand side, we have

$$\begin{aligned} & prod \circ F\ return \\ = \quad & \{ \text{datatype isomorphism} \} \\ & prod \circ F\ return \circ (Wrap \triangledown Return) \circ out_F \\ = \quad & \{ \text{naturality of } Wrap \triangledown Return \} \\ & prod \circ (Wrap \triangledown Return) \circ (M\ F\ return + return) \circ out_F \\ = \quad & \{ \text{definitions of } prod_1, prod_2 \} \\ & (prod_1 \triangledown prod_2) \circ (M\ F\ return + return) \circ out_F \\ = \quad & \{ \text{coproducts} \} \\ & ((prod_1 \circ M\ F\ return) \triangledown (prod_2 \circ return)) \circ out_F \end{aligned}$$

and for the right-hand side we have

$$\begin{aligned} & return_M \\ = \quad & \{ \text{datatype isomorphism} \} \\ & return_M \circ (Wrap \triangledown Return) \circ out_F \\ = \quad & \{ \text{coproducts} \} \\ & ((return_M \circ Wrap) \triangledown (return_M \circ Return)) \circ out_F \end{aligned}$$

These two expressions must be equal, which implies that the equalities

$$\begin{aligned} prod_1 \circ M\ F\ return &= return_M \circ Wrap \\ prod_2 \circ return &= return_M \circ Return \end{aligned}$$

must hold. The second equality follows from $prod_2 = id$ and the definition of *return*; we'll use the first one to derive a definition for $prod_1$. We have:

$$\begin{aligned} & prod_1 \circ M\ F\ return \\ = \quad & \{ \text{(A) suppose that } prod_1 = prod'_1 \circ M\ prod \} \\ & prod'_1 \circ M\ prod \circ M\ F\ return \\ = \quad & \{ \text{functors} \} \end{aligned}$$

$$
\begin{aligned}
& \mathit{prod}_1' \circ M\ (\mathit{prod} \circ F\ \mathit{return}) \\
= & \quad \{\ \text{(B) induction}\ \} \\
& \mathit{prod}_1' \circ M\ \mathit{return}_M \\
= & \quad \{\ \text{(C) suppose that } \mathit{prod}_1' = \mathit{return}_M \circ \mathit{Wrap} \circ \mathit{join}_M\ \} \\
& \mathit{return}_M \circ \mathit{Wrap} \circ \mathit{join}_M \circ M\ \mathit{return}_M \\
= & \quad \{\ M \text{ is a monad}\ \} \\
& \mathit{return}_M \circ \mathit{Wrap}
\end{aligned}
$$

and so property (2) strongly suggests letting

$$\mathit{prod}_1 = \mathit{return}_M \circ \mathit{Wrap} \circ \mathit{join}_M \circ M\ \mathit{prod}$$

The three steps (A), (B), (C) need a little justification. For (A), we're heading towards a use of induction, so we require an occurrence of *M prod* at the end of $\mathit{prod}_1$. For (B), induction seems plausible, because this calculation takes places under a *Wrap* constructor. For (C), the final $M\ \mathit{return}_M$ is cancellable via $\mathit{join}_M$, so we're done—we let be $\mathit{prod}_1'$ be the final expression we want composed with this cancellation.

We don't need any stronger justification for induction than mere plausibility, because we are using this calculation only to suggest a possible definition of $\mathit{prod}_1$ that we should then check more rigorously. This still leaves us also with having to check property (3), again by induction. Both of these proofs are presented in Appendix A.

The final instance declaration for *Nest* is then as follows:

$$
\begin{aligned}
& \textbf{instance}\ (\mathit{Functor}\ m,\ \mathit{Monad}\ m) \Rightarrow \mathit{Monad}\ (\mathit{Nest}\ m)\ \textbf{where} \\
& \quad \mathit{return} = \mathit{Nest} \circ \mathit{return}_M \circ \mathit{Return} \\
& \quad \mathit{join} = \mathit{Nest} \circ \mathit{join}_M \circ \mathit{fmap}_M\ \mathit{prod} \circ \mathit{unNest} \\
& \qquad \textbf{where} \\
& \qquad\quad \mathit{prod}\ (\mathit{Return}\ (\mathit{Nest}\ m)) = m \\
& \qquad\quad \mathit{prod}\ (\mathit{Wrap}\ m) \qquad\quad\ = (\mathit{return}_M \circ \mathit{Wrap} \circ \mathit{join}_M \circ \mathit{fmap}_M\ \mathit{prod})\ m
\end{aligned}
$$

### 3.3 Tracing with *Nest*

For any monad *m*, the monad *Nest m* is a tracer. The *lift* function maps *Return* on values. It does not create any *Wrap* constructor, so lifted computations are single-level and are always joined when *join* for *Nest* is performed. The *drop* function traverses the free structure, and collapses levels which were previously separated by *Wrap* constructors. The *mark* gadget explicitly creates a new level by the *Wrap* constructor mapped on *Return*s. All future computations will be confined to the wrapped *Return*s, and so the previous structure is preserved.

$$
\begin{aligned}
& \textbf{instance}\ (\mathit{Functor}\ m,\ \mathit{Monad}\ m) \Rightarrow \mathit{MonadTrans}\ m\ (\mathit{Nest}\ m)\ \textbf{where} \\
& \quad \mathit{lift} \quad\ = \mathit{Nest} \circ \mathit{fmap}\ \mathit{Return} \\
& \quad \mathit{drop}\ v = \mathit{unNest}\ v \gg\!= \mathit{revert} \\
& \qquad \textbf{where} \\
& \qquad\quad \mathit{revert}\ (\mathit{Return}\ a) = \mathit{return}\ a \\
& \qquad\quad \mathit{revert}\ (\mathit{Wrap}\ m) \ \ = m \gg\!= \mathit{revert} \\
& \textbf{instance}\ (\mathit{Functor}\ m,\ \mathit{Monad}\ m) \Rightarrow \mathit{MonadTrace}\ (\mathit{Nest}\ m)\ \textbf{where} \\
& \quad \mathit{mark} \ \ = (\mathit{Nest} \circ \mathit{return} \circ \mathit{Wrap} \circ \mathit{return} \circ \mathit{Return})\ ()
\end{aligned}
$$

The proof that these definitions satisfy the laws of *MonadTrace* from Section 3.1 is given in Appendix B.

As an example of *mark*, consider the following conversation with the Haskell shell. The call of the *mark* operation in the second query maps *Wrap* on every element. This way, every future computation is confined to the inside of those two *Wrap* constructors. Whatever monadic computation is bound to the result, on the top-level there will always be a two-element list, because it is a node with only *Wrap* constructors as children.

$\triangleright$ *lift* $[1,2]$
*Nest* [*Return* 1, *Return* 2]
$\triangleright$ **do** $\{x \leftarrow$ *lift* $[1,2]$; *lift* $[0\,..\,x]\}$
*Nest* [*Return* 0, *Return* 1, *Return* 0, *Return* 1, *Return* 2]
$\triangleright$ **do** $\{$*lift* $[1,2]$; *mark*$\}$
*Nest* [*Wrap* [*Return* ()], *Wrap* [*Return* ()]]
$\triangleright$ **do** $\{x \leftarrow$ *lift* $[1,2]$; *mark*; *lift* $[0\,..\,x]\}$
*Nest* [*Wrap* [*Return* 0, *Return* 1], *Wrap* [*Return* 0, *Return* 1, *Return* 2]]

# 4 Interpreting traces

The tracing construction has multiple applications. As in the motivating example from the introduction, we can track the course of computation, in order to identify any point of failure. But, stepping back from this particular example, we observe that a trace is simply a data structure, and the computation in the *Nest* monad is performed only if the data structure is forced. This gives us control over the process of execution, which can be useful in the context of non-terminating computations (although the inductive proofs will need strengthening in that context). We can interpret the free parts of a *Nest* value in any way we want, and thus mix in some new effects, not previously available with the original monad. This control over a computation from the outside strongly resembles the paradigm of aspect-oriented programming.

## 4.1 The partiality monad

Capretta introduced the partiality monad [3] to capture non-termination as an effect; this technique has applications in type theory, to model non-guarded recursion. The original formulation is as follows.

**data** *Partial a* = *Later* (*Partial a*) | *Now a*

A structure of this type is a value wrapped in a (possibly infinite) number of *Later* constructors. It represents a computation sliced into layers. We can explicitly force any number of layers. The *Nest* monad can be seen as a monad transformer which allows computations structured by any monad to be sliced. The most basic case, *Nest Identity*, is indeed isomorphic to *Partial*.

In most programming languages, including Haskell, the $\vee$ operator is asymmetric—non-strict in its second argument ($True \vee \bot \equiv True$) but strict in its first ($\bot \vee True \equiv \bot$). In pure Haskell, it is impossible to define *parallel-or*—that is, a disjunction $\curlyvee$ with the property that $True \curlyvee \bot \equiv True \equiv \bot \curlyvee True$.

To implement such a disjunction in the real world, we need some kind of parallelism, so that both arguments are evaluated simultaneously; when either one terminates with *True* or both terminate with *False*, the computation of the disjunction is complete. We can purely approximate this behaviour—at least for the case where undefined arguments arise from non-termination, rather than from any other reason. We do this by explicitly cutting the infinite computation into finite pieces, using the *Nest* tracer.

Consider the following function, which is an implementation of the so-called Collatz problem. It is suspected that for all $n > 0$ it returns *True*, but no proof for this claim is known.

$$\begin{array}{l}
\mathit{collatz} :: \mathit{Integer} \to \mathit{Bool} \\
\mathit{collatz}\ 1 \qquad\qquad\quad = \mathit{True} \\
\mathit{collatz}\ n \mid \mathit{odd}\ n \qquad = \mathit{collatz}\ (3 \times n + 1) \\
\qquad\qquad \mid \mathit{otherwise} = \mathit{collatz}\ (n \div 2)
\end{array}$$

If we want to check whether at least one of two Collatz sequences ends, it is not the best idea to use the regular Haskell disjunction, since if the first one diverges, the whole function diverges too.

$$\begin{array}{l}
\mathit{oneOf} :: \mathit{Integer} \to \mathit{Integer} \to \mathit{Bool} \\
\mathit{oneOf}\ a\ b = \mathit{collatz}\ a \lor \mathit{collatz}\ b
\end{array}$$

A much safer solution is to chop the evaluation of the Collatz sequence into pieces. We use the *Nest* tracer for the *Identity* monad. This way, we make the evaluation incremental, which enables us to execute it in parallel. The scheduler is very simple and hidden in the definition of $\curlyvee$. It executes a piece from each argument in turn.

$$\begin{array}{l}
\mathit{collatzN} :: \mathit{Integer} \to \mathit{Nest}\ \mathit{Identity}\ \mathit{Bool} \\
\mathit{collatzN}\ 1 \qquad\qquad\quad = \mathit{return}\quad \mathit{True} \\
\mathit{collatzN}\ n \mid \mathit{odd}\ n \qquad = \mathit{mark} \gg \mathit{collatzN}\ (3 \times n + 1) \\
\qquad\qquad\quad \mid \mathit{otherwise} = \mathit{mark} \gg \mathit{collatzN}\ (n \div 2)
\end{array}$$

$$\begin{array}{llll}
(\curlyvee) :: \mathit{Nest}\ \mathit{Identity}\ \mathit{Bool} \to \mathit{Nest}\ \mathit{Identity}\ \mathit{Bool} \to \mathit{Bool} & & & \\
\mathit{Nest}\ (\mathit{Identity}\ (\mathit{Leaf}\ \mathit{False})) & \curlyvee\ \mathit{Nest}\ (\mathit{Identity}\ (\mathit{Leaf}\ \mathit{False})) & = \mathit{False} \\
\mathit{Nest}\ (\mathit{Identity}\ (\mathit{Leaf}\ \mathit{True})) & \curlyvee\ _ & = \mathit{True} \\
_ & \curlyvee\ \mathit{Nest}\ (\mathit{Identity}\ (\mathit{Leaf}\ \mathit{True})) & = \mathit{True} \\
\mathit{Nest}\ (\mathit{Identity}\ (\mathit{Node}\ m)) & \curlyvee\ x & = x \curlyvee \mathit{Nest}\ m
\end{array}$$

We can test the $\curlyvee$ operator as follows. Note that *collatz* diverges if applied to 0.

$$\begin{array}{l}
\triangleright\ \mathit{collatzN}\ 120 \curlyvee \mathit{collatzN}\ 130 \\
\mathit{True} \\
\triangleright\ \mathit{collatzN}\ 0 \curlyvee \mathit{collatzN}\ 130 \\
\mathit{True} \\
\triangleright\ \mathit{collatzN}\ 130 \curlyvee \mathit{collatzN}\ 0 \\
\mathit{True}
\end{array}$$

This model can easily be extended to different kinds of such thread races. For example, it is possible to simulate McCarthy's ambiguous choice operator $\mathit{amb} :: a \to b \to \mathit{Either}\ a\ b$ [18], which has the property that for $a_0 :: a$ and $b_0 :: b$, $\mathit{amb}\ a_0 \perp = \mathit{Left}\ a_0$, and $\mathit{amb} \perp b_0 = \mathit{Right}\ b_0$ (again, assuming that the undefined values arise from non-termination).

## 4.2 Approximating computations

For some monads, the initial segment of a *Nest* value may be seen as an "approximation" to the computation. Consider the monad of finitely supported probability distributions. Its most common representation is a list of probabilities paired with values.

$$
\begin{array}{l}
\textbf{newtype}\ \mathit{Distr}\ a = \mathit{Distr}\{\ \mathit{runDistr} :: [(\mathit{Double}, a)]\ \} \\
\textbf{instance}\ \mathit{Functor}\ \mathit{Distr}\ \textbf{where} \\
\quad \mathit{fmap}\ f\ (\mathit{Distr}\ \mathit{xs}) = \mathit{Distr}\ (\mathit{fmap}\ (\lambda (p,a) \to (p, f\ a))\ \mathit{xs}) \\
\textbf{instance}\ \mathit{Monad}\ \mathit{Distr}\ \textbf{where} \\
\quad \mathit{return}\ a = \mathit{Distr}\ [(1,a)] \\
\quad \mathit{join}\ (\mathit{Distr}\ \mathit{xs}) = \mathit{Distr}\ [(p \times q, a) \mid (p, \mathit{Distr}\ \mathit{ys}) \leftarrow \mathit{xs},\ (q,a) \leftarrow \mathit{ys}]
\end{array}
$$

However, this representation has a flaw. Consider the following problem: given the uniform distribution of $\{0,1\}$ (a fair coin), select uniformly an element from $\{0,1,2\}$. A solution is to flip the coin twice to get the uniform distribution of $\{0,1,2,3\}$; if you draw 0, 1, or 2, this is your answer, and if you draw 3, flip twice more. In Haskell:

$$
\begin{array}{l}
\mathit{coin}\ :: \mathit{Distr}\ \mathit{Int} \\
\mathit{coin}\ = \mathit{Distr}\ [(0.5,0),(0.5,1)] \\
\mathit{third} :: \mathit{Distr}\ \mathit{Int} \\
\mathit{third} = \textbf{do}\ x \leftarrow \mathit{coin} \\
\qquad\qquad\quad y \leftarrow \mathit{coin} \\
\qquad\qquad\quad \textbf{case}\ (x,y)\ \textbf{of} \\
\qquad\qquad\qquad (1,1) \to \mathit{return}\ 0 \\
\qquad\qquad\qquad (1,0) \to \mathit{return}\ 1 \\
\qquad\qquad\qquad (0,1) \to \mathit{return}\ 2 \\
\qquad\qquad\qquad (0,0) \to \mathit{third}
\end{array}
$$

(This is a simplification of Knuth and Yao's technique to simulate a fair die using three fair coin tosses [15].) Though the solution is mathematically reasonable, the Haskell implementation is useless, because the *List* monad, and so also the *Distr* monad, gathers the results in a depth-first fashion. Though this may not be obvious at first sight, the recursive call in *third* is in the head of the list. Therefore, *third* actually diverges without producing any usable results: $\mathit{third} = \perp$.

The *Nest* monad can retrieve the situation, if we suspend the recursive call.

$$
\begin{array}{l}
\mathit{thirdN} :: \mathit{Nest}\ \mathit{Distr}\ \mathit{Int} \\
\mathit{thirdN} = \textbf{do}\ x \leftarrow \mathit{lift}\ \mathit{coin} \\
\qquad\qquad\quad\ \ y \leftarrow \mathit{lift}\ \mathit{coin} \\
\qquad\qquad\quad\ \ \textbf{case}\ (x,y)\ \textbf{of} \\
\qquad\qquad\qquad\ \ (1,1) \to \mathit{return}\ 0 \\
\qquad\qquad\qquad\ \ (1,0) \to \mathit{return}\ 1 \\
\qquad\qquad\qquad\ \ (0,1) \to \mathit{return}\ 2 \\
\qquad\qquad\qquad\ \ (0,0) \to \mathit{mark} \gg \mathit{thirdN}
\end{array}
$$

What can we do with *thirdN*? One possibility is to get an "approximation" of the structure with the following function, which cuts the subcomputations if the recursion is deeper than the specified argument. All the computations that are too deep are replaced with *Nothing*.

$$
\begin{array}{l}
\mathit{takeN} :: (\mathit{Functor}\ m,\ \mathit{Monad}\ m) \Rightarrow \mathit{Int} \to \mathit{Nest}\ m\ a \to \mathit{Nest}\ m\ (\mathit{Maybe}\ a) \\
\mathit{takeN}\ k\ (\mathit{Nest}\ m) = \mathit{Nest}\ (\mathit{fmap}\ (\mathit{aux}\ k)\ m) \\
\quad \mathbf{where} \\
\qquad \mathit{aux}\ 0\ (\mathit{Wrap}\ \_\,) \ \ = \mathit{Return}\ \mathit{Nothing} \\
\qquad \mathit{aux}\ k\ (\mathit{Wrap}\ m) \ \ = \mathit{Wrap}\ (\mathit{fmap}\ (\mathit{aux}\ (k-1))\ m) \\
\qquad \mathit{aux}\ k\ (\mathit{Return}\ a) = \mathit{Return}\ (\mathit{Just}\ a)
\end{array}
$$

$$
\begin{array}{l}
\mathit{approx} :: (\mathit{Functor}\ m,\ \mathit{Monad}\ m) \Rightarrow \mathit{Int} \to \mathit{Nest}\ m\ a \to m\ (\mathit{Maybe}\ a) \\
\mathit{approx}\ k = \mathit{drop} \circ \mathit{takeN}\ k
\end{array}
$$

We can ask the Haskell shell:

$$
\begin{array}{l}
\triangleright\ \mathit{approx}\ 0\ \mathit{thirdN} \\
[\,(0.25,\mathit{Nothing}),(0.25,\mathit{Just}\ 2),(0.25,\mathit{Just}\ 1),(0.25,\mathit{Just}\ 0)\,] \\
\triangleright\ \mathbf{let}\ \mathit{simpl}\ (\mathit{Distr}\ \mathit{xs}) = \mathit{Distr}\ (\mathit{map}\ (\lambda x \to (\mathit{sum}\ [p \mid (p,a) \leftarrow \mathit{xs}, x \equiv a], x))\ (\mathit{nub}\ (\mathit{fmap}\ \mathit{snd}\ \mathit{xs}))) \\
\triangleright\ \mathit{simpl}\ (\mathit{approx}\ 1\ \mathit{thirdN}) \\
[\,(0.0625,\mathit{Nothing}),(0.3125,\mathit{Just}\ 2),(0.3125,\mathit{Just}\ 1),(0.3125,\mathit{Just}\ 0)\,] \\
\triangleright\ \mathit{simpl}\ (\mathit{approx}\ 2\ \mathit{thirdN}) \\
[\,(0.015635,\mathit{Nothing}),(0.32813,\mathit{Just}\ 2),(0.32813,\mathit{Just}\ 1),(0.32813,\mathit{Just}\ 0)\,] \\
\triangleright\ \mathit{simpl}\ (\mathit{approx}\ 10\ \mathit{thirdN}) \\
[\,(2.38419\mathrm{e}{-7},\mathit{Nothing}),(0.33333,\mathit{Just}\ 2),(0.33333,\mathit{Just}\ 1),(0.33333,\mathit{Just}\ 0)\,]
\end{array}
$$

### 4.3 The Prolog *cut* operator

Prolog's *cut* operator allows one to restrict backtracking. The moment it is reached during evaluation of a predicate, it succeeds, but also discards all the possible backtrack choices created by the predicate so far. By exposing the structure of a *List* computation, we can use this effect also in Haskell. We perform a depth-first search on a rose tree of type *Nest* [ ], but once we go down a level, we never go back.

$$
\begin{array}{l}
\mathit{call} :: \mathit{Nest}\ [\,]\ a \to [a] \\
\mathit{call}\ (\mathit{Nest}\ \mathit{xs}) = \mathit{aux}\ \mathit{xs} \\
\quad \mathbf{where} \\
\qquad \mathit{aux}\ [\,] \qquad\qquad\quad = [\,] \\
\qquad \mathit{aux}\ (\mathit{Return}\ a : \mathit{xs}) = a : \mathit{aux}\ \mathit{xs} \\
\qquad \mathit{aux}\ (\mathit{Wrap}\ \mathit{as} : \_\,) = \mathit{aux}\ \mathit{as}
\end{array}
$$

$$
\begin{array}{l}
\mathit{brace} :: \mathit{Nest}\ [\,]\ a \to \mathit{Nest}\ [\,]\ a \\
\mathit{brace} = \mathit{lift} \circ \mathit{call}
\end{array}
$$

$$
\begin{array}{l}
\mathit{cut} :: \mathit{Nest}\ [\,]\ () \\
\mathit{cut} = \mathit{mark}
\end{array}
$$

Consider the following example.

$$
\begin{array}{l}
\triangleright\ \mathit{call}\ (\ \mathbf{do}\ x \leftarrow \mathit{lift}\ [4,7,13,9] \\
\qquad\qquad\quad y \leftarrow \mathit{lift}\ [2,8,1] \\
\qquad\qquad\quad \mathit{when}\ (x+y \geqslant 15)\ \mathit{cut} \\
\qquad\qquad\quad \mathit{return}\ (x+y)\ ) \\
[6,12,5,9,15]
\end{array}
$$

(Here, *when* is a standard Haskell function, defined by $when\ b\ m = \mathbf{if}\ b\ \mathbf{then}\ m\ \mathbf{else}\ return\ ()$.) First, we pick 4 from the first list, which after choices from the second list creates $[6,12,5]$. Then, we pick 7 from the first list. We cut on the second element of the second choice (because $7+8=15$). So, all the other choices from the second list (that is, 1) are discarded, as well as other choices from the first list (that is, 13 and 9).

We can limit the scope of *cut* by using the *brace* function. Only choices from inside of the brace are now cut.

$$
\begin{array}{l}
\triangleright\ call\ (\ \mathbf{do}\ x \leftarrow lift\ [4,7,13,9] \\
\qquad\qquad\quad brace\ (\ \mathbf{do}\ y \leftarrow lift\ [2,8,1] \\
\qquad\qquad\qquad\qquad\qquad when\ (x+y \geqslant 15)\ cut \\
\qquad\qquad\qquad\qquad\qquad return\ (x+y)\ )\ ) \\
[6,12,5,9,15,15,11,17]
\end{array}
$$

Different interpretations of the *Nest* data structure enable the definition of different search strategies, such as breadth-first search. Moreover, it can even mix two different strategies in lifted and minded parts.

## 4.4 Poor man's concurrency transformer, revisited

Claessen's "poor man's" concurrency transformer [5] adds simple concurrency capabilities to any monad. It has two flaws. The first one is that it does not respect the laws presented in Section 3. Every lifted operation is atomic, and execution can only be interrupted in between atomic actions; this means that the evaluation of $lift\ m_1 \gg lift\ m_2$ can be interrupted by an action from another thread, while $lift\ (m_1 \gg m_2)$ cannot. The second flaw is that the return type of its *run* function is $m\ ()$, and so it does not allow one to collect actual results of the computation. Here, we give a version of Claessen's transformer which fixes these flaws, via a conscious use of free structures.

In Claessen's monad, the user first builds a continuation, which produces an expression, which then is interpreted by the *run* function. By augmenting the type of expressions, we can skip the continuation layer. The datatype for concurrent expressions is as follows.

$$\mathbf{data}\ Action\ m\ a = Par\ (Action\ m\ a)\ (Action\ m\ a) \mid Act\ (m\ (Action\ m\ a)) \mid Done\ a \mid Kill$$

Intuitively, the *Par* constructor pairs two expressions for parallel evaluation, *Act* performs a single monadic action, *Done* terminates the computation with an answer[1], and *Kill* terminates the computation with no answer. We treat *Action* as a term algebra with *Done* as a constructor for variables.

$$
\begin{array}{l}
\mathbf{instance}\ Functor\ m \Rightarrow Monad\ (Action\ m)\ \mathbf{where} \\
\quad \begin{array}{lll}
return & & = Done \\
Par\ a\ b & \gg\!= f & = Par\ (a \gg\!= f)\ (b \gg\!= f) \\
Act\ m & \gg\!= f & = Act\ (fmap\ (\gg\!= f)\ m) \\
Done\ a & \gg\!= f & = f\ a \\
Kill & \gg\!= f & = Kill
\end{array}
\end{array}
$$

The *Action* datatype describes a program, but does not specify which actions form atomic chunks that should not be interrupted by other operations. This task is delegated to the *Nest* transformer. The

[1] The *Kill* constructor is called *Stop* in Claessen's datatype, and *Done* is new.

type of our concurrent monad is then *Nest* (*Action m*) *a*, for a monad *m* and answer type *a*. We define a number of operations, which allow easy construction of concurrent expressions. The functions *done* and *kill* lift the appropriate constructors. A single operation can be embedded in an *Action* data structure and lifted to the concurrent monad with *act*. There are two operators for concurrency, *par* and *fork*. The former constructs a computation from two computations of the same type. The latter starts an auxiliary thread, whose final value is ignored (see the examples below).

$$
\begin{array}{l}
\textbf{type}\ \mathit{Concurrent}\ m = \mathit{Nest}\ (\mathit{Action}\ m)\\[4pt]
\mathit{done} :: (\mathit{Monad}\ m) \Rightarrow a \rightarrow \mathit{Concurrent}\ m\ a\\
\mathit{done} = \mathit{lift} \circ \mathit{Done}\\[4pt]
\mathit{kill} :: (\mathit{Monad}\ m) \Rightarrow \mathit{Concurrent}\ m\ a\\
\mathit{kill} = \mathit{lift}\ \mathit{Kill}\\[4pt]
\mathit{act} :: (\mathit{Monad}\ m) \Rightarrow m\ a \rightarrow \mathit{Concurrent}\ m\ a\\
\mathit{act}\ m = \mathit{lift}\ (\mathit{Act}\ (\mathit{liftM}\ \mathit{Done}\ m))\\[4pt]
\mathit{par} :: (\mathit{Monad}\ m) \Rightarrow \mathit{Concurrent}\ m\ a \rightarrow \mathit{Concurrent}\ m\ a \rightarrow \mathit{Concurrent}\ m\ a\\
\mathit{par}\ (\mathit{Nest}\ m_1)\ (\mathit{Nest}\ m_2) = \mathit{Nest}\ (\mathit{Par}\ (\mathit{Done}\ (\mathit{Wrap}\ m_1))\ (\mathit{Done}\ (\mathit{Wrap}\ m_2)))\\[4pt]
\mathit{fork} :: (\mathit{Monad}\ m) \Rightarrow \mathit{Concurrent}\ m\ b \rightarrow \mathit{Concurrent}\ m\ ()\\
\mathit{fork}\ m = \mathit{par}\ (m \gg \mathit{kill})\ (\mathit{act}\ (\mathit{return}\ ()))
\end{array}
$$

We schedule such computations with the following *round* function. We can see *Done* as a constructor which either terminates evaluation of an atomic chunk (when it is composed with *Wrap*) or the entire thread (when it is composed with *Return*).

$$
\begin{array}{l}
\mathit{round} :: \mathit{Monad}\ m \Rightarrow [\mathit{Nest}\ (\mathit{Action}\ m)\ x] \rightarrow m\ [x]\\
\mathit{round}\ [\,] \qquad\qquad\quad = \mathit{return}\ [\,]\\
\mathit{round}\ (\mathit{Nest}\ w : \mathit{as}) = \textbf{case}\ w\ \textbf{of}\\
\quad \mathit{Kill} \qquad\qquad\quad\ \rightarrow \mathit{round}\ \mathit{as}\\
\quad \mathit{Done}\ (\mathit{Return}\ x) \rightarrow \textbf{do}\ \{\mathit{xs} \leftarrow \mathit{round}\ \mathit{as};\ \mathit{return}\ (x : \mathit{xs})\}\\
\quad \mathit{Done}\ (\mathit{Wrap}\ a) \quad \rightarrow \mathit{round}\ (\mathit{as} +\!\!+ [\mathit{Nest}\ a])\\
\quad \mathit{Act}\ m \qquad\qquad\ \rightarrow \textbf{do}\ \{a \leftarrow m;\ \mathit{round}\ ([\mathit{Nest}\ a] +\!\!+ \mathit{as})\}\\
\quad \mathit{Par}\ a\ b \qquad\quad\ \ \rightarrow \mathit{round}\ ([\mathit{Nest}\ b] +\!\!+ \mathit{as} +\!\!+ [\mathit{Nest}\ a])
\end{array}
$$

We can test our monad as follows. In the first example, we first define two expressions: *cat* writes the string `"cat"` five times, relinquishing control every time the operation is performed. Similarly, *fish* writes `"fish"` seven times.

$$
\begin{array}{l}
\textbf{instance}\ (\mathit{Monoid}\ s) \Rightarrow \mathit{MonadWriter}\ s\ (\mathit{Concurrent}\ (\mathit{Writer}\ s))\ \textbf{where}\\
\quad \mathit{tell} = \mathit{act} \circ \mathit{tell}\\[4pt]
\mathit{cat} :: \mathit{Concurrent}\ (\mathit{Writer}\ \mathit{String})\ \mathit{Int}\\
\mathit{cat} = \mathit{replicateM}\ 5\ (\mathit{tell}\ \texttt{"cat"} \gg \mathit{mark}) \gg \mathit{return}\ 1\\[4pt]
\mathit{fish} :: \mathit{Concurrent}\ (\mathit{Writer}\ \mathit{String})\ \mathit{Int}\\
\mathit{fish} = \mathit{replicateM}\ 7\ (\mathit{tell}\ \texttt{"fish"} \gg \mathit{mark}) \gg \mathit{return}\ 2
\end{array}
$$

We can test them, by running them in parallel.

$$
\begin{array}{l}
\triangleright\ \mathit{round}\ [\textbf{do}\ x \leftarrow \mathit{fish}\ \text{`}\mathit{par}\text{`}\ \mathit{cat}\\
\qquad\qquad\quad \mathit{tell}\ \texttt{"dog"}
\end{array}
$$

$$
\begin{array}{l}
\qquad\qquad \mathit{return}\ x] \\
(\texttt{"catfishcatfishcatfishcatfishcatfishdogfishfishdog"}, [1,2])
\end{array}
$$

The results of all parallel threads called with *par*, in this example $[1,2]$, are returned in a list. The operation *tell* `"dog"` is bound to both threads.

We can now run *fish* in a separate, auxiliary thread. The thread is run on the side, the following actions are not bound to it, and its result is not returned with the overall result.

$$
\begin{array}{l}
\triangleright\ \mathit{round}\ [\mathbf{do}\ \mathit{fork}\ \mathit{fish} \\
\qquad\qquad x \leftarrow \mathit{cat} \\
\qquad\qquad \mathit{tell}\ \texttt{"dog"} \\
\qquad\qquad \mathit{return}\ x] \\
(\texttt{"catfishcatfishcatfishcatfishcatfishdogfishfish"}, [1])
\end{array}
$$

We can also define a version of *fish* that is performed atomically.

$$
\begin{array}{l}
\mathit{fish}' :: \mathit{Concurrent}\ (\mathit{Writer}\ \mathit{String})\ \mathit{Int} \\
\mathit{fish}' = \mathit{replicateM}\ 7\ (\mathit{tell}\ \texttt{"fish"}) \gg \mathit{return}\ 2
\end{array}
$$

We can see that *round* does not separate the calls of *tell* `"fish"`.

$$
\begin{array}{l}
\triangleright\ \mathit{round}\ [\mathbf{do}\ \mathit{fork}\ \mathit{fish}' \\
\qquad\qquad x \leftarrow \mathit{cat} \\
\qquad\qquad \mathit{tell}\ \texttt{"dog"} \\
\qquad\qquad \mathit{return}\ x] \\
(\texttt{"catfishfishfishfishfishfishfishcatcatcatcatdog"}, [1])
\end{array}
$$

## 5 Related work

The idea of separation of syntax and semantics of monadic computations is not new. It is the very foundation of the success of monads as a tool for encapsulating impure behaviour in pure languages [21]—for example, the way Haskell integrates impure effects such as I/O within a pure language is to make the pure evaluation *construct* a syntactic term, which is subsequently *interpreted* by the run-time system.

The work most related to ours is the Unimo framework introduced by Lin [17], which is an embedded domain-specific language designed to modularise construction of monads in Haskell. Even though Lin's motivation and toolbox significantly differ from ours, he came to the same conclusion that exposing the structure of computations allows more functionality to be added to existing monads. We share a strong flavour of aspect-oriented programming.

There is a resemblance between resumptions, used for modelling semantics of concurrency [8, 11], and the *Nest* monad. A *resumption monad transformer* is used, for example, by Papaspyrou to model semantics of concurrency in domain theory [20], and by Harrison in his "cheap" concurrency [9, 10]. They both use a definition similar to ours from Section 2 (though they fail to mention the connection between free monads and resumptions), and as a result their constructions are not transformers in the sense of Section 3.

Harrison [9] is not quite right in claiming that Claessen's monad is based on first-class continuations. A version of the continuation monad is used only to build a syntactic term, which serves as the backbone of the concurrent computation. The term is not a monad, since it lacks free variables, but it reveals the structure of resumptions (notice the type of the constructor $Atom :: m\ (Action\ m) \to Action\ m$). Similarly, in Swierstra and Altenkirch's functional specification of concurrency [23], resumptions are used implicitly, and control is surrendered by a thread whenever it wants to communicate with other parts of the concurrent system (which is denoted by a constructor of the free structure $IO_c$). As we show in Sections 4.1 and 4.4, the *Nest* monad transformer allows one to separate the concepts of composition of computations and yielding, by an explicit use of *mark*.

A definition of a resumption transformer, which satisfies the laws from Section 3, and is in fact isomorphic to *Nest*, was already given by Cenciarelli and Moggi [4], but practical applications in programming were not studied. Also—as pointed to us by the anonymous reviewers—the *Nest* transformer can be obtained via Hyland, Plotkin and Power's *sum* construction [13], as a sum of a monad and an *Identity*-generated free monad. Though Hyland *et al.*'s construction provides a simpler proof that *Nest* can be given a monad structure (using a distributive law between a monad *m* and the *m*-generated free monad), our definition of *join* is not intensionally similar to the one arising from the *sum* construction, and issues like efficiency should also be taken into account. A naive implementation of *join* in the sum construction traverses the structure twice (once to apply the distributive law, and once to join the free structure), while *join* for *Nest* needs to traverse the structure only once. On the other hand, the *sum* construction allows one to include an additional functor—the datatype in question is of the form $M\ (Free\ (F \circ M)\ a)$—which may help to generalize our notion of tracing in the future.

The interleaving between pure data and monadic structure was also considered by Filinski and Støvring [7], and in forthcoming work by Atkey *et al.* [1]. They give proof principles for reasoning about datatypes that include effects, for example a stream in which tails are always guarded by I/O actions.

## 6 Future work

So far, we failed to mention the mother and father of all purely functional monads, that is the continuation and state monads. We do not have much to say about continuations, but we see a lot of applications in tracing the *State* monad.

**Functional specifications of effects.** The idea behind functional specifications of effects in pure languages is to model the logic of an effectful construct in the pure core of the language. For example, such a specification may consist of a datatype representing a model of the outside world and a variation of the *State* monad whose state modifications mirror the actions of the side-effecting monad. This way, we can translate a program into its pure equivalent, test it, and reason about it no differently from how we would reason about any other pure program.

The existing frameworks for specifying "effectful" Haskell in "pure" Haskell—like those proposed by Swierstra and Altenkirch [22, 23] and Butterfield *et al.* [2, 6]—do not concentrate on non-terminating computations, which are of little use in the pure world, but which are back in the spotlight in the presence of effects like I/O and concurrency. For example, Butterfield *et al.* model the interaction between programs and a filesystem by means of the *State* monad, which for an initial state (of the filesystem) produces a final value and a final state. In case of non-terminating programs, no final state exists, so the whole model becomes useless. What we are really interested in is not a final state, but the whole (possibly infinite) sequence of subsequent states of the filesystem, or a trace of all the interactions. In such a setting, we can use coinduction as a reasoning tool, which coincides with the non-strict semantics of Haskell, in which we try to embed our model.

As mentioned before, our approach can transform monolithic computations into coinductive unfolding of traces. That is why we propose to use a different monad as a basis for functional specifications. It is a monad which produces not only the final state, but the whole (possibly infinite) stream of intermediate states.

$$\textbf{data}\ \mathit{Trace}\ s\ a = \mathit{TCons}\ s\ (\mathit{Trace}\ s\ a) \mid \mathit{Nil}\ a$$
$$\textbf{newtype}\ \mathit{States}\ s\ a = \mathit{States}\{\mathit{runStates} :: s \to \mathit{Trace}\ s\ a\}$$

We leave the exact implementation of instances of *Monad* $(\mathit{States}\ s)$, *MonadTrans* $(\mathit{State}\ s)$ $(\mathit{States}\ s)$ and *MonadTrace* $(\mathit{States}\ s)$ to the reader as an exercise. The *mark* operation should accumulate the current state in the *Trace*.

What is the relationship between *States s* and *Nest* $(\mathit{State}\ s)$? It is possible to interpret free parts of *Nest* $(\mathit{State}\ s)$ in a suitable way, that is to define a monad morphism of type $\mathit{Nest}\ (\mathit{State}\ s) \to \mathit{States}\ s$. We can also suspect a different kind of generality, since *State* is a composition of two adjoint functors, namely $\mathit{State}\ s = \mathit{Reader}\ s \circ \mathit{Writer}\ s$, while $\mathit{States}\ s = \mathit{Reader}\ s \circ \mathit{Free}\ (\mathit{Writer}\ s)$.

## References

[1] Robert Atkey, Neil Ghani, Bart Jacobs & Patricia Johann (2012): *Fibrational Induction Meets Effects*. In Lars Birkedal, editor: *Foundations of Software Science and Computational Structures*, *Lecture Notes in Computer Science* 7213, Springer, pp. 42–57. To Appear.

[2] Andrew Butterfield & Glenn Strong (2002): *Proving Correctness of Programs with I/O - A Paradigm Comparison*. In: *Selected Papers from the 13th International Workshop on Implementation of Functional Languages*, IFL '02, Springer-Verlag, London, UK, pp. 72–87, doi:10.1007/3-540-46028-4_5.

[3] Venanzio Capretta (2005): *General Recursion via Coinductive Types*. *Logical Methods in Computer Science* 1(2), pp. 1–18, doi:10.2168/LMCS-1(2:1)2005.

[4] Pietro Cenciarelli & Eugenio Moggi (1993): *A Syntactic Approach to Modularity in Denotational Semantics*. In: *Proceedings of the 5th Biennial Meeting on Category Theory and Computer Science*, CTCS '93, CWI Technical Report, Amsterdam, The Netherlands.

[5] Koen Claessen (1999): *A poor man's concurrency monad*. *Journal of Functional Programming* 9, pp. 313–323, doi:10.1017/S0956796899003342.

[6] Malcolm Dowse, Glenn Strong & Andrew Butterfield (2003): *Proving make correct: I/O proofs in Haskell and Clean*. In: *Proceedings of the 14th International Conference on Implementation of Functional Languages*, IFL'02, Springer-Verlag, Berlin, Heidelberg, pp. 68–83, doi:10.1007/3-540-44854-3_5.

[7] Andrzej Filinski & Kristian Støvring (2007): *Inductive reasoning about effectful data types*. In: *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming*, ICFP '07, ACM, New York, NY, USA, pp. 97–110, doi:10.1145/1291151.1291168.

[8] Steven E. Ganz, Daniel P. Friedman & Mitchell Wand (1999): *Trampolined style*. In: *Proceedings of the fourth ACM SIGPLAN International Conference on Functional Programming*, ICFP '99, ACM, New York, NY, USA, pp. 18–27, doi:10.1145/317636.317779.

[9] William L. Harrison: *Cheap (But Functional) Threads*. Available at `http://www.cs.missouri.edu/~harrison/drafts/CheapThreads.pdf`.

[10] William L. Harrison (2006): *The Essence of Multitasking*. In: *1st International Conference on Algebraic Methodology and Software Technology (AMAST06)*, *LNCS* 4019, Springer-Verlag, doi:10.1007/11784180_14.

[11] Matthew Hennessy & Gordon Plotkin (1979): *Full Abstraction for a Simple Parallel Programming Language*. In Jirí Becvár, editor: *MFCS*, *Lecture Notes in Computer Science* 74, Springer, pp. 108–120, doi:10.1007/3-540-09526-8_8.

[12] Ralf Hinze (2000): *Deriving backtracking monad transformers*. In: *Proceedings of the fifth ACM SIGPLAN International Conference on Functional Programming*, ICFP '00, ACM, New York, NY, USA, pp. 186–197, doi:10.1145/351240.351258.

[13] Martin Hyland, Gordon Plotkin & John Power (2006): *Combining effects: sum and tensor*. *Theoretical Computer Science* 357, pp. 70–99, doi:10.1016/j.tcs.2006.03.013.

[14] Mark P. Jones & Luc Duponcheel (1993): *Composing Monads*. Technical Report, Yale University.

[15] Donald Knuth & Andrew Yao (1976): *The Complexity of Nonuniform Random Number Generation*. In J. F. Traub, editor: *Algorithms and Complexity: New Directions and Recent Results*, Academic Press. Also in *Selected Papers on the Analysis of Algorithms*.

[16] Sheng Liang, Paul Hudak & Mark Jones (1995): *Monad transformers and modular interpreters*. In: *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '95, ACM, New York, NY, USA, pp. 333–343, doi:10.1145/199448.199528.

[17] Chuan-kai Lin (2006): *Programming monads operationally with Unimo*. In: *Proceedings of the eleventh ACM SIGPLAN International Conference on Functional Programming*, ICFP '06, ACM, New York, NY, USA, pp. 274–285, doi:10.1145/1159803.1159840.

[18] John McCarthy (1963): *A Basis for a Mathematical Theory of Computation*. In P. Braffort & D. Hirschberg, editors: *Computer Programming and Formal Systems*, North-Holland, Amsterdam, pp. 33–70, doi:10.1016/S0049-237X(08)72018-4.

[19] Eugenio Moggi (1991): *Notions of computation and monads*. *Information and Computation* 93, pp. 55–92, doi:10.1016/0890-5401(91)90052-4.

[20] Nikolaos S. Papaspyrou (2001): *A Resumption Monad Transformer and its Applications in the Semantics of Concurrency*. In: *Proceedings of the 3rd Panhellenic Logic Symposium*, Anogia, Crete.

[21] Simon L. Peyton Jones & Philip Wadler (1993): *Imperative functional programming*. In: *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '93, ACM, New York, NY, USA, pp. 71–84, doi:10.1145/158511.158524.

[22] Wouter Swierstra (2008): *A Functional Specification of Effects*. Ph.D. thesis, University of Nottingham.

[23] Wouter Swierstra & Thorsten Altenkirch (2007): *Beauty in the beast*. In: *Proceedings of the ACM SIGPLAN Workshop on Haskell*, Haskell '07, ACM, New York, NY, USA, pp. 25–36, doi:10.1145/1291201.1291206.

# Appendix A Proof that *Nest* is a monad

We prove the properties (2) and (3) from Section 3.2 by induction, assuming an initial-algebra reading of the datatype *Free m*—that is, we assume a well-founded ordering on the subterms of any value of type *Free m a*. (We have to resort to something like induction, because the definition of *prod* above isn't in the form of a standard recurson pattern—in particular, it is not a fold.) For brevity, we omit the *Nest* constructor.

Property (2): The case for *Return* is straightforward. For the *Wrap* case, we assume that the property holds for each element of data structure *m*; that is, that

$$M\ (prod \circ F\ return)\ m = M\ return_M\ m$$

Then we calculate:

$$\begin{array}{ll}
& (prod \circ F\ return)\ (Wrap\ m) \\
= & \quad \{ \text{ naturality of } Wrap :: M\ F \to F \ \} \\
& (prod \circ Wrap \circ M\ F\ return)\ m \\
= & \quad \{ \text{ definition of } prod \ \} \\
& (return_M \circ Wrap \circ join_M \circ M\ prod \circ M\ F\ return)\ m \\
= & \quad \{ \text{ functors } \} \\
& (return_M \circ Wrap \circ join_M \circ M\ (prod \circ F\ return))\ m \\
= & \quad \{ \text{ induction } \} \\
& (return_M \circ Wrap \circ join_M \circ M\ return_M)\ m \\
= & \quad \{ \ M \text{ as a monad } \} \\
& (return_M \circ Wrap)\ m
\end{array}$$

Property (3): The case for *Return* is again straightforward. In the *Wrap* case, we again assume that the property holds for each element of *m*:

$$M\ (join \circ prod)\ m = M\ (prod \circ F\ join)\ m$$

Then we calculate:

$$\begin{array}{ll}
& (join \circ prod)\ (Wrap\ m) \\
= & \quad \{ \text{ definition of } join \ \} \\
& (join_M \circ M\ prod \circ prod \circ Wrap)\ m \\
= & \quad \{ \text{ definition of } prod \ \} \\
& (join_M \circ M\ prod \circ return_M \circ Wrap \circ join_M \circ M\ prod)\ m \\
= & \quad \{ \text{ naturality of } return_M \ \} \\
& (join_M \circ return_M \circ prod \circ Wrap \circ join_M \circ M\ prod)\ m \\
= & \quad \{ \ M \text{ as a monad } \} \\
& (prod \circ Wrap \circ join_M \circ M\ prod)\ m \\
= & \quad \{ \text{ definition of } prod \ \} \\
& (return_M \circ Wrap \circ join_M \circ M\ prod \circ join_M \circ M\ prod)\ m \\
= & \quad \{ \text{ naturality of } join_M \ \} \\
& (return_M \circ Wrap \circ join_M \circ join_M \circ M\ M\ prod \circ M\ prod)\ m \\
= & \quad \{ \ M \text{ as a monad } \} \\
& (return_M \circ Wrap \circ join_M \circ M\ join_M \circ M\ M\ prod \circ M\ prod)\ m
\end{array}$$

$$
\begin{array}{ll}
= & \{ \text{functors} \} \\
& (\mathit{return}_M \circ \mathit{Wrap} \circ \mathit{join}_M \circ M\ (\mathit{join}_M \circ M\ \mathit{prod} \circ \mathit{prod}))\ m \\
= & \{ \text{definition of } \mathit{join}\text{; induction} \} \\
& (\mathit{return}_M \circ \mathit{Wrap} \circ \mathit{join}_M \circ M\ (\mathit{prod} \circ F\ \mathit{join}))\ m \\
= & \{ \text{functors} \} \\
& (\mathit{return}_M \circ \mathit{Wrap} \circ \mathit{join}_M \circ M\ \mathit{prod} \circ M\ F\ \mathit{join})\ m \\
= & \{ \text{definition of } \mathit{join} \} \\
& (\mathit{return}_M \circ \mathit{Wrap} \circ \mathit{join} \circ M\ F\ \mathit{join})\ m \\
= & \{ \text{definition of } \mathit{prod} \} \\
& (\mathit{prod} \circ \mathit{Wrap} \circ M\ F\ \mathit{join})\ m \\
= & \{ \text{naturality of } \mathit{Wrap} :: M\ F \mathrel{\dot{\to}} F \} \\
& (\mathit{prod} \circ F\ \mathit{join})\ (\mathit{Wrap}\ m)
\end{array}
$$

## Appendix B Proof that *Nest* is a tracer

Here, we prove that *Nest* is a tracer (see Sections 3.1 and 3.3 for the definitions). The equalities $\mathit{drop} \circ \mathit{lift} = \mathit{id}$ and $\mathit{drop}\ \mathit{mark} = \mathit{return}\ ()$ are straightforward.

We prove the equality $\mathit{lift} \circ \mathit{return}_M = \mathit{return}_N$ as follows.

$$
\begin{array}{ll}
& \mathit{lift} \circ \mathit{return}_M \\
= & \{ \text{definition of } \mathit{lift} \} \\
& M\ \mathit{Return} \circ \mathit{return}_M \\
= & \{ \text{naturality of } \mathit{return}_M \} \\
& \mathit{return}_M \circ \mathit{Return} \\
= & \{ \text{definition of } \mathit{return}_N \} \\
& \mathit{return}_N
\end{array}
$$

The fact that $\mathit{lift}\ c \gg\!\!=_N \mathit{lift} \circ f = \mathit{lift}\ (c \gg\!\!=_M f)$ follows from the following.

$$
\begin{array}{ll}
& \mathit{lift}\ c \gg\!\!=_N \mathit{lift} \circ f \\
= & \{ \text{definition of } \gg\!\!=_N \} \\
& \mathit{join}_N\ (N\ (\mathit{lift} \circ f)\ (\mathit{lift}\ c)) \\
= & \{ \text{definition of } \mathit{lift} \} \\
& \mathit{join}_N\ (N\ (M\ \mathit{Return} \circ f)\ (M\ \mathit{Return}\ c)) \\
= & \{ \text{definition of } N \} \\
& \mathit{join}_N\ (MF\ (M\ \mathit{Return} \circ f)\ (M\ \mathit{Return}\ c)) \\
= & \{ \text{definition of } \mathit{join}_N \} \\
& \mathit{join}_M\ (M\ \mathit{prod}\ (MF\ (M\ \mathit{Return} \circ f)\ (M\ \mathit{Return}\ c))) \\
= & \{ \text{functor} \} \\
& \mathit{join}_M\ (M\ (\mathit{prod} \circ F\ (M\ \mathit{Return} \circ f) \circ \mathit{Return})\ c) \\
= & \{ \text{definition of } F \} \\
& \mathit{join}_M\ (M\ (\mathit{prod} \circ \mathit{Return} \circ M\ \mathit{Return} \circ f)\ c) \\
= & \{ \text{definition of } \mathit{prod} \} \\
& \mathit{join}_M\ (M\ (M\ \mathit{Return} \circ f)\ c) \\
= & \{ \text{naturality of } \mathit{join}_M \}
\end{array}
$$

$$
\begin{array}{ll}
& (M\ \mathit{Return} \circ \mathit{join}_M)\ (M\ f\ c) \\
= & \{\ \text{definition of } \mathit{lift}\ \} \\
& \mathit{lift}\ (\mathit{join}_M\ (M\ f\ c)) \\
= & \{\ \text{definition of } \mathit{join}_M\ \} \\
& \mathit{lift}\ (c \gg\!\!=_M f)
\end{array}
$$

To prove the fact that *drop* is a monad morphism we first prove a lemma

$$\mathit{join}_M \circ M\ \mathit{revert} \circ \mathit{prod} \circ F\ f = \mathit{join}_M \circ \mathit{join}_M \circ MM\ \mathit{revert} \circ M\ f \circ \mathit{revert}$$

The case for *Return* follows from simple unfolding of the definitions. The case for *Wrap* is as follows.

$$
\begin{array}{ll}
& (\mathit{join}_M \circ M\ \mathit{revert} \circ \mathit{prod} \circ F\ f)\ (\mathit{Wrap}\ m) \\
= & \{\ \text{definition of } F\ \} \\
& (\mathit{join}_M \circ M\ \mathit{revert} \circ \mathit{prod})\ (\mathit{Wrap}\ (MF\ f)\ m)) \\
= & \{\ \text{definition of } \mathit{prod}\ \} \\
& (\mathit{join}_M \circ M\ \mathit{revert} \circ \mathit{return}_M \circ \mathit{Wrap} \circ \mathit{join}_M \circ M\ \mathit{prod} \circ MF\ f)\ m \\
= & \{\ \text{naturality of } \mathit{return}_M\ \} \\
& (\mathit{join}_M \circ \mathit{return}_M \circ \mathit{revert} \circ \mathit{Wrap} \circ \mathit{join}_M \circ M\ \mathit{prod} \circ MF\ f)\ m \\
= & \{\ \text{monad laws}\ \} \\
& (\mathit{revert} \circ \mathit{Wrap} \circ \mathit{join}_M \circ M\ \mathit{prod} \circ MF\ f)\ m \\
= & \{\ \text{definition of } \mathit{revert}\ \} \\
& (\mathit{join}_M \circ M\ \mathit{revert} \circ \mathit{join}_M \circ M\ \mathit{prod} \circ MF\ f)\ m \\
= & \{\ \text{naturality of } \mathit{join}_M\ \} \\
& (\mathit{join}_M \circ \mathit{join}_M \circ MM\ \mathit{revert} \circ M\ \mathit{prod} \circ MF\ f)\ m \\
= & \{\ \text{monad laws}\ \} \\
& (\mathit{join}_M \circ M\ \mathit{join}_M \circ MM\ \mathit{revert} \circ M\ \mathit{prod} \circ MF\ f))\ m \\
= & \{\ \text{functor}\ \} \\
& (\mathit{join}_M \circ M\ (\mathit{join}_M \circ M\ \mathit{revert} \circ \mathit{prod} \circ F\ f))\ m \\
= & \{\ \text{induction}\ \} \\
& (\mathit{join}_M \circ M\ (\mathit{join}_M \circ M\ \mathit{join}_M \circ MM\ \mathit{revert} \circ M\ f \circ \mathit{revert})\ m \\
= & \{\ \text{functor}\ \} \\
& (\mathit{join}_M \circ M\ \mathit{join}_M \circ MM\ \mathit{join}_M \circ MMM\ \mathit{revert} \circ MM\ f \circ M\ \mathit{revert})\ m \\
= & \{\ \text{monad laws}\ \} \\
& (\mathit{join}_M \circ \mathit{join}_M \circ MM\ \mathit{join}_M \circ MMM\ \mathit{revert} \circ MM\ f \circ M\ \mathit{revert})\ m \\
= & \{\ \text{naturality of } \mathit{join}_M\ \} \\
& (\mathit{join}_M \circ M\ \mathit{join}_M \circ \mathit{join\_m} \circ MMM\ \mathit{revert} \circ MM\ f \circ M\ \mathit{revert})\ m \\
= & \{\ \text{naturality of } \mathit{join}_M\ \} \\
& (\mathit{join}_M \circ M\ \mathit{join}_M \circ MM\ \mathit{revert} \circ \mathit{join}_M \circ MM\ f \circ M\ \mathit{revert})\ m \\
= & \{\ \text{naturality of } \mathit{join}_M\ \} \\
& (\mathit{join}_M \circ \mathit{join}_M \circ MM\ \mathit{revert} \circ M\ f \circ \mathit{join}_M \circ M\ \mathit{revert})\ m \\
= & \{\ \text{definition of } \mathit{revert}\ \} \\
& (\mathit{join}_M \circ \mathit{join}_M \circ MM\ \mathit{revert} \circ M\ f \circ \mathit{revert})\ (\mathit{Wrap}\ m)
\end{array}
$$

To prove that $\mathit{drop}\ (c \gg\!\!=_N f) = \mathit{drop}\ c \gg\!\!=_M (\mathit{drop} \circ f)$, for $c :: a$ and $f :: a \to \mathit{Nest}\ m\ b$, we unfold the definition of $(\gg\!\!=)$ and prove the equivalent equality $\mathit{drop}\ (\mathit{join}_N\ (N\ f\ c)) = \mathit{join}_M\ (M\ (\mathit{drop} \circ f)\ \mathit{drop}\ c)$. It follows from the commutativity of the following diagram (in Hask). For brevity, we write $\mu_N$ for the *join* of monad *Nest*. The left-most path of the diagram is equal to $join_M \circ M\ (drop \circ f) \circ drop$, while the right-most path is equal to $drop \circ join_N \circ N\,f$.

$$
\begin{tikzcd}
MFA \arrow[r, equal] \arrow[d, "M\ revert"] \arrow[dd, bend right, "drop"'] & MFA \arrow[r, equal] \arrow[d, "M\ revert"] & MFA \arrow[r, equal] \arrow[d, "MF\,f"] & MFA \arrow[d, "MF\,f"'] \\
MMA \arrow[r, equal] \arrow[d, "\mu_M"] & MMA \arrow[d, "MM\,f"] & MFMFB \arrow[r, equal] \arrow[d, "M\ prod"] & MFMFB \arrow[d, "M\ prod"'] \arrow[dd, bend left, "\mu_N"] \\
MA \arrow[d, "M\,f"] & MMMFB \arrow[d, "MMM\ revert"] & MMFB \arrow[r, equal] \arrow[d, "MM\ revert"] & MMFB \arrow[d, "\mu_M\ F"'] \\
MMFB \arrow[d, "MM\ revert"] \arrow[dd, bend right, "M\ drop"'] & MMMMB \arrow[d, bend right, "MM\ \mu_M"'] \arrow[d, bend left, "M\ \mu_M\ M"] & MMMB \arrow[dd, bend right, "M\ \mu_M"'] \arrow[dd, bend left, "\mu_M\ M"] & MFB \arrow[dd, "M\ revert"'] \arrow[ddd, bend left, "drop"] \\
MMMB \arrow[d, "M\ \mu_M"] & MMMB \arrow[d, bend right, "\mu_M\ M"'] \arrow[d, bend left, "M\ \mu_M"] & & \\
MMB \arrow[d, "\mu_M"'] & MMB \arrow[r, equal] \arrow[d, "\mu_M"] & MMB \arrow[r, equal] \arrow[d, "\mu_M"] & MMB \arrow[d, "\mu_M"'] \\
MB \arrow[r, equal] & MB \arrow[r, equal] & MB \arrow[r, equal] & MB
\end{tikzcd}
$$

The first column of the diagram depicts unfolding of the definitions of *drop*. The second column is equal to the first one due to the fact that $join_M$ is a natural transformation: the morphism $join_M :: MMA \to MA$ form the first column "travelled" down the path to settle as $join_M :: MMMB \to MMB$. Additionally, due to the monad law for $join_M$, we can exchange $MM\ join_M$ with $M\ join_M$. This allows us to use the lemma, the $M$-image of which commutes columns 2 and 3. Again, we can use the monad law to change $M\ join_M$ into $join_M$. We use the naturality of $join_M$ to switch its position with the mapping of *revert*, which justify the fourth column.