Update Monads: Cointerpreting Directed Containers*

Danel Ahman[1] **and Tarmo Uustalu**[2]

1 **Laboratory for Foundations of Computer Science, University of Edinburgh**
10 Crichton Street, Edinburgh EH8 9AB, United Kingdom
d.ahman@ed.ac.uk

2 **Institute of Cybernetics at Tallinn University of Technology**
Akadeemia tee 21, 12618 Tallinn, Estonia
tarmo@cs.ioc.ee

Abstract

We introduce update monads as a generalization of state monads. Update monads are the compatible compositions of reader and writer monads given by a set and a monoid. Distributive laws between such monads are given by actions of the monoid on the set.

We also discuss a dependently typed generalization of update monads. Unlike simple update monads, they cannot be factored into a reader and writer monad, but rather into similarly looking relative monads.

Dependently typed update monads arise from cointerpreting directed containers, by which we mean an extension of an interpretation of the opposite of the category of containers into the category of set functors.

1998 ACM Subject Classification F.3.2 Semantics of Programming Languages, F.3.3 Studies of Program Constructs

Keywords and phrases monads and distributive laws, reader and writer and state monads, monoids, monoid actions, directed containers

Digital Object Identifier 10.4230/LIPIcs.TYPES.2013.1

1 Introduction

In denotational semantics and functional programming, reader, writer and state monads [15] are well known and important. They are related to each other, but there is also something that may feel unsatisfactory: reader and writer monads are not instances of state monads and state monads are not combinations of reader and writer monads.

In this paper we introduce a generalization of state monads, which we call update monads, that overcome exactly this underachievement. Update monads are compatible compositions of reader and writer monads, they are specified by a set, a monoid and an action, defining a reader monad, a writer monad and a distributive law of the latter over the former. They collect computations that take an initial state to pair of an update (that is not applied!) and a return value.

We also discuss a dependently typed generalization of update monads. Dependently typed update monads arise from cointerpreting directed containers, by which we mean an extension

* This work was supported by the University of Edinburgh Principal's Career Development PhD Scholarship, the ERDF funded Estonian CoE project EXCS and ICT programme project Coinduction, the Estonian Ministry of Education and Research target-financed research theme no. 0140007s12 and the Estonian Science Foundation grant no. 9475.

of an interpretation of the opposite of the category of containers into the category of set functors. Directed containers [2] are a description of comonoids in the category of containers and characterize those containers whose interpretation carries a comonad structure. In a directed container, each state has its own set of updates that are considered "safe" for that state.

The paper is organized as follows. We begin Section 2 by recapitulating reader, writer and state monads. We then introduce update monads, show that reader and writer monads are instances of update monads and that state monads related to them in an important way; we also discuss algebras of update monads. Next, we show that update monads are compatible compositions of reader and writer monads, which leads to a different characterization of their algebras. We also briefly discuss update monad maps. In Section 3, we look at the dependently typed generalization of update monads that results from cointerpreting directed containers. In Section 4, we compare update monads to a generalization of state monads by Kammar and Plotkin.

For self-containedness of the paper, in Appendix A, we review monoids, actions, monads and compatible compositions of monads. In Appendix B, we give a detailed proof of the main theorem of the paper. In Appendix C, we show how algebras of update monads can be described as models of Lawvere theories.

In this paper we develop the theory of update monads over the category **Set** of sets and functions. The development in Section 2 can be easily generalized to arbitrary Cartesian closed categories. The development in Section 3 can be similarly carried out in locally Cartesian closed categories.

2 Unifying reader, writer, state monads

2.1 Reader, writer, state monads

We recall the three "classic" families of monads of functional programming [15]—the reader, writer and state monads.

Reader monads

Every set S (of states) defines a monad (the *reader monad*) via

$$T\,X = S \to X$$

$$\eta : \forall\{X\}.\, X \to S \to X$$
$$\eta\, x = \lambda s.\, x$$

$$\mu : \forall\{X\}.\, (S \to (S \to X)) \to S \to X$$
$$\mu\, f = \lambda s.\, f\, s\, s$$

Here and in the following, we use Agda's [12] syntax of braces for implicit arguments, i.e., for those arguments we may want to skip when they are inferrable from other arguments.

Writer monads

Every monoid $(P, \mathsf{o}, \oplus)$ (of updates) defines a monad (the *writer monad*, also sometimes called the *complexity* monad) via

$$T\,X = P \times X$$

$$\eta : \forall\{X\}.\, X \to P \times X$$
$$\eta\, x = (\mathsf{o}, x)$$

$$\mu : \forall\{X\}.\, P \times (P \times X) \to P \times X$$
$$\mu\,(p, (p', x)) = (p \oplus p', x)$$

Note that we would not be able to do with just a set P to get a monad on this underlying functor, we need both the unit and multiplication of the monoid to the define the unit and multiplication and the monoid laws to prove the monad laws.[1]

State monads

Every set S defines a monad (the *state monad*) via

$$T\,X = S \to S \times X$$

$$\eta : \forall\{X\}.\, X \to S \to S \times X$$
$$\eta\, x = \lambda s.\, (s, x)$$

$$\mu : \forall\{X\}.\, (S \to S \times (S \to S \times X)) \to S \to S \times X$$
$$\begin{aligned}\mu\, f = \lambda s.\, &\mathsf{let}\ (s', g) = f\ s;\\ &\quad\ (s'', x) = g\ s'\\ &\mathsf{in}\ (s'', x)\end{aligned}$$

The usual explanation of the state monad is the following. $T\,X = S \to S \times X$ is the set of computations each taking an initial state to a pair of a final state and a return value.

Unifying the three?

Notice the similarities between these monads. The reader monad resembles the state monad, when we ignore the final, mutated, state. The writer monad resembles the state monad, when we ignore the initial state and require the final state to have monoidal structure. (This said, in the case of the writer monad, the values written accumulate, but in the case of the state monad, they replace each other.) Could we possibly unify the three properly? We can. We do this in the next section using monoid actions.

2.2 Update monads

Given a set S and a monoid $(P, \mathsf{o}, \oplus)$ together with a right action $\downarrow\colon S \times P \to S$ of the monoid on the set (in this paper we call such a triple a *(right) act*[2]), we are interested in the

[1] The monoid structure on P induces also a different non-isomorphic monad defined by $T^{\mathrm{r}}X = X \times P$, $\eta^{\mathrm{r}}\, x = (x, \mathsf{o})$, $\mu^{\mathrm{r}}\,((x, p), p') = (x, p \oplus p')$. The order of P and X in the product $T^{\mathrm{r}}\,X$ is not important here, as product is symmetric. But μ^{r} adds the two given monoid elements in the order reverse to μ.

[2] A set that a fixed monoid $(P, \mathsf{o}, \oplus)$ acts on is often called a $(P, \mathsf{o}, \oplus)$-set or a $(P, \mathsf{o}, \oplus)$-act. We are interested in varying both the set S and the monoid $(P, \mathsf{o}, \oplus)$ at the same time.

following monad, which we call the *update monad* for the act $(S, (P, \mathsf{o}, \oplus), \downarrow)$.

$$T\,X = S \to P \times X$$

$$\eta : \forall\{X\}.\, X \to S \to P \times X$$
$$\eta\, x = \lambda s.\, (\mathsf{o}, x)$$

$$\mu : \forall\{X\}.\, (S \to P \times (S \to P \times X)) \to S \to P \times X$$
$$\begin{aligned}\mu\, f = \lambda s.\, &\mathsf{let}\ (p, g) = f\, s;\\ &\quad\ (p', x) = g\, (s \downarrow p)\\ &\mathsf{in}\ (p \oplus p', x)\end{aligned}$$

A computation over X, i.e., an element of $T\,X$, is a function taking an initial state to an update produced and a return value. Notice that rather than returning the result of applying the update to the initial state, i.e., the final state, the function returns the actual update itself. These updates are only ever applied by the multiplication μ. This operation applies to the initial state s the update p defined by s, in order to thus obtain a new state $s \downarrow p$ that then further determines a new update p' to be composed with p.

▶ **Example 1.** It turns out that reader and writer monads are special cases of update monads.

We get the reader monad for a given set S when we take $(P, \mathsf{o}, \oplus)$ and $\downarrow$ trivial, i.e., $P = 1$. We then have $T\,X = S \to 1 \times X \cong S \to X$.

The writer monad for a given monoid $(P, \mathsf{o}, \oplus)$ is obtained by taking S and $\downarrow$ trivial, i.e., $S = 1$, so that $T\,X = 1 \to P \times X \cong P \times X$.

▶ **Example 2.** State monads fail to be a special case of update monads, but they are very close in an important way.

Recall that the free monoid on any semigroup[3] $(S, \bullet)$ is $(P, \mathsf{o}, \oplus)$ where

$$P = 1 + S$$

$$\mathsf{o} = \mathsf{inl}\ *$$

$$\begin{aligned}&\mathsf{inl}\ * \oplus p = p\\ &\mathsf{inr}\ s \oplus \mathsf{inl}\ * = \mathsf{inr}\ s\\ &\mathsf{inr}\ s \oplus \mathsf{inr}\ s' = \mathsf{inr}\ (s \bullet s')\end{aligned}$$

The actions of this monoid on any set S are determined by the actions of the inducing semigroup $(S, \bullet)$ on S. Recall that, in particular, $\bullet$ is an action of $(S, \bullet)$ on S, so it also induces an action of $(P, \mathsf{o}, \oplus)$ on S.

Notice also that, for any set S, the "overwrite" operation $\bullet$ defined by $s \bullet s' = s'$ gives a semigroup structure on S.[4]

Now let us fix some set S and let (T, η, μ) be the state monad for S. Let $(P, \mathsf{o}, \oplus)$ be the free monoid on the overwrite semigroup $(S, \bullet)$ and let $\downarrow$ be the action of $(P, \mathsf{o}, \oplus)$ on S induced by $\bullet$. Let $(T^\circ, \eta^\circ, \mu^\circ)$ be the update monad for S, $(P, \mathsf{o}, \oplus)$ and $\downarrow$.

It turns out that the state monad (T, η, μ) is characterized as the splitting of the following monad idempotent *idem* on $(T^\circ, \eta^\circ, \mu^\circ)$ that replaces the nil update with overwriting the

[3] Notice that we are talking about the free monoid on a semigroup here, not the free monoid on a set! This is about adjoining a unit element to the semigroup.

[4] In semigroup/monoid literature [10], this is called the right zero semigroup structure.

given state by itself:

$$\begin{aligned}&\mathit{idem} : \forall\{X\}.\, (S \to (1 + S) \times X) \to S \to (1 + S) \times X\\ &\mathit{idem}\, f = \lambda s.\, \mathsf{let}\ (p, x) = f\, s\ \mathsf{in}\ (\mathsf{inr}\ (\mathsf{case}\ p\ \mathsf{of}\ (\mathsf{inl} * \mapsto s; \mathsf{inr}\, s' \mapsto s')), x)\end{aligned}$$

Indeed the monad (T, η, μ) embeds into $(T^\circ, \eta^\circ, \mu^\circ)$ via a monad map sec:

$$\begin{aligned}&\mathit{sec} : \forall\{X\}.\, (S \to S \times X) \to S \to (1 + S) \times X\\ &\mathit{sec}\, f = \lambda s.\, \mathsf{let}\ (s', x) = f\, s\ \mathsf{in}\ (\mathsf{inr}\, s', x)\end{aligned}$$

The monad $(T^\circ, \eta^\circ, \mu^\circ)$ also projects onto the state monad (T, η, μ) via a monad map retr:

$$\begin{aligned}&\mathit{retr} : \forall\{X\}.\, (S \to (1 + S) \times X) \to S \to S \times X\\ &\mathit{retr}\, f = \lambda s.\, \mathsf{let}\ (p, x) = f\, s\ \mathsf{in}\ (\mathsf{case}\ p\ \mathsf{of}\ (\mathsf{inl} * \mapsto s; \mathsf{inr}\, s' \mapsto s'), x)\end{aligned}$$

And it is easy to check that $\mathit{sec} \circ \mathit{retr} = \mathit{idem}$ and $\mathit{retr} \circ \mathit{sec} = \mathsf{id}$.

▶ **Example 3.** We mentioned earlier that the functor $T_\circ$ given by $T_\circ\, X = P_\circ \times X$ is not a monad, if $P_\circ$ is just a set. We need the unit and multiplication of a monoid structure on $P_\circ$ for $T_\circ$ to carry the unit and multiplication of a monad.

But in fact on any set $P_\circ$ we have the the overwrite semigroup structure. Hence $T_\circ$ is at least a "monad without a unit" with multiplication $\mu_\circ : P_\circ \times (P_\circ \times X) \to P_\circ \times X$ given by $\mu_\circ\, (p, (p', x)) = (p', x)$.

We get a monad from $(T_\circ, \mu_\circ)$ by freely adjoining a unit. Concretely, we get a monad (T, η, μ) by defining

$$T\, X = X + P_\circ \times X \quad (\cong (1 + P_\circ) \times X)$$

$$\begin{aligned}&\eta : \forall\{X\}.\, X \to X + P_\circ \times X\\ &\eta\, x = \mathsf{inl}\, x\end{aligned}$$

$$\begin{aligned}&\mu : \forall\{X\}.\, (X + P_\circ \times X) + P_\circ \times (X + P_\circ \times X) \to X + P_\circ \times X\\ &\mu\, (\mathsf{inl}\, c) = c\\ &\mu\, (\mathsf{inr}\, (p, (\mathsf{inl}\, x))) = \mathsf{inr}\, (p, x)\\ &\mu\, (\mathsf{inr}\, (p, (\mathsf{inr}\, (p', x)))) = \mathsf{inr}\, (p', x)\end{aligned}$$

This is (up to isomorphism) the update monad for the set 1, the free monoid on the overwrite semigroup on $P_\circ$ and the trivial action. We call it the *overwrite monad*.

▶ **Example 4.** For any set S and monoid $(P, \mathsf{o}, \oplus)$, we have an action that does nothing: $s \downarrow p = s$. The multiplication operation of the corresponding update monad uses the same state twice:

$$T\, X = S \to P \times X$$

$$\begin{aligned}&\eta : \forall\{X\}.\, X \to S \to P \times X\\ &\eta\, x = \lambda s.\, (\mathsf{o}, x)\end{aligned}$$

$$\begin{aligned}&\mu : \forall\{X\}.\, (S \to P \times (S \to P \times X)) \to S \to P \times X\\ &\mu\, f = \lambda s.\, \mathsf{let}\ (p, g) = f\, s;\\ &\qquad\qquad (p', x) = g\, s\\ &\qquad\ \mathsf{in}\ (p \oplus p', x)\end{aligned}$$

▶ **Example 5.** Here is a cute example of update monads of with a clear programming meaning.

Let $(P, \mathsf{o}, \oplus)$ be the free monoid on a given set S explicitly defined by

$$P = S^*$$

$$\mathsf{o} = []$$
$$ss \oplus ss' = ss +\!\!+ ss'$$

i.e., the set of lists over S, the empty list and concatenation. As the action $\downarrow : S \to S^* \to S$ we want to use

$$s \downarrow ss = \mathsf{last}\,(s :: ss)$$

(Note that :: can be given the type $S \times S^* \to S^+$ and last is a total function $S^+ \to S$.)

We get the following *state-logging* monad (similar to the one considered by Piróg and Gibbons [13], except that it only allows finite traces):

$$T\,X = S \to S^* \times X$$

$$\eta : \forall\{X\}.\, X \to S \to S^* \times X$$
$$\eta\, x = \lambda s.\, ([], x)$$

$$\mu : \forall\{X\}.\, (S \to S^* \times (S \to S^* \times X)) \to S \to S^* \times X$$
$$\begin{aligned}\mu\, f = \lambda s.\, &\mathsf{let}\ (ss, g) = f\, s;\\ &\quad\ (ss', x) = g\,(\mathsf{last}\,(s :: ss))\\ &\mathsf{in}\ (ss +\!\!+ ss', x)\end{aligned}$$

A computation takes an initial state to the list of all intermediate states (excluding the initial state!) and the value returned.

Here and in the following we use Haskell notation for lists, with [] for nil, :: for cons and $+\!\!+$ for append. In addition we will write $es\rfloor n$ for taking n first elements of a list es, $n\lfloor es$ for taking n last elements, and es/n for removing n last elements of es (if $\mathsf{len}\, es \leq n$, then all elements are taken resp. removed).

▶ **Example 6.** Here is a minimally more involved concrete example of an update monad—for no-removal *buffers* of a fixed size N. This is the definition:

$$T\,X = E^{\leq N} \to E^* \times X$$

$$\eta : \forall\{X\}.\, X \to E^{\leq N} \to E^* \times X$$
$$\eta\, x = \lambda es.\, ([], x)$$

$$\mu : \forall\{X\}.\, (E^{\leq N} \to E^* \times (E^{\leq N} \to E^* \times X)) \to E^{\leq N} \to E^* \times X$$
$$\begin{aligned}\mu\, f = \lambda es.\, &\mathsf{let}\ (es', g) = f\, es;\\ &\quad\ (es'', x) = g\,(es +\!\!+ (es'\rfloor(N - \mathsf{len}\, es)))\\ &\mathsf{in}\ (es' +\!\!+ es'', x)\end{aligned}$$

The buffer is used to store values drawn from some given set E and has size N. Therefore, we take as the states $S = E^{\leq N}$ lists of values of length at most N (for values stored in the buffer). The updates $P = E^*$ are simply lists of values to write into the buffer, with the nil update and composition of two updates given by $\mathsf{o} = []$, $es \oplus es' = es +\!\!+ es'$. The action, defined by $es \downarrow es' = es +\!\!+ (es'\rfloor(N - \mathsf{len}\, es))$, updates the buffer with

additional values, as long as there is free space. The updates that do not fully fit into the buffer are performed partially, so some suffix of the list of values to write may be dropped "silently". (An alternative buffer might prefer new values to old, which corresponds to choosing $es \downarrow es' = N\lfloor(es +\!\!+ es')$.)

▶ **Example 7.** To implement an unbounded stack, we can choose the set of states to be $S = E^*$ (values stored in the stack) and as the set of updates use $P = (1 + E)^*$, $\mathsf{o} = []$, $ps \oplus ps' = ps +\!\!+ ps'$ (sequences of pop and push instructions). The intended action $\downarrow$ is then

$$
\begin{aligned}
&es \downarrow [] = es \\
&es \downarrow (\mathsf{inl} * :: ps) = es/1 \downarrow ps \\
&es \downarrow (\mathsf{inr}\, e :: ps) = (es +\!\!+ [e]) \downarrow ps
\end{aligned}
$$

(notice that popping from the empty stack removes no element).

Alternatively, we can be more abstract in regards to updates and identify all those sequences of pop and push instructions that have the same net effect. An update is then a number of elements to remove from the stack and a list of new elements to add. We define

$$
\begin{aligned}
&P' = \mathsf{Nat} \times E^* \\
&\mathsf{o}' = (0, []) \\
&(n, es) \oplus' (n', es') = (n + (n' \dot{-} \mathsf{len}\, es), es/n' +\!\!+ es')
\end{aligned}
$$

$$
es \downarrow' (n', es') = es/n' +\!\!+ es'
$$

The monoid here is a Zappa-Szép product [5] of the monoids $(\mathsf{Nat}, 0, +)$ and $(E^*, [], +\!\!+)$. It arises from two matching actions of the two monoids on each other.

In Section 3, we will see that a dependently typed version of update monads can disallow over- and underflowing updates.

2.3 Algebras of update monads

By the definition of an algebra of a monad (see Appendix A.2), an algebra for the update monad for an act $(S, (P, \mathsf{o}, \oplus), \downarrow)$ is a set X with an operation

$$
act : (S \to P \times X) \to X
$$

satisfying the equations

$$
\begin{aligned}
&x = act\, (\lambda s.\, (\mathsf{o}, x)) \\
&act\, (\lambda s.\, (p\, s, act\, (\lambda s'.\, (p'\, s\, s', x\, s\, s')))) = act\, (\lambda s.\, (p\, s \oplus p'\, s\, (s \downarrow p\, s), x\, s\, (s \downarrow p\, s)))
\end{aligned}
$$

However it is quite easy to see that the same thing can also be described as a set X with two operations (see the interdefinability below)

$$
\begin{aligned}
&lkp : (S \to X) \to X \\
&upd : P \times X \to X
\end{aligned}
$$

satisfying the equations

$$
\begin{aligned}
&x = lkp\, (\lambda s.\, upd\, (\mathsf{o}, x)) \\
&upd\, (p, upd\, (p', x)) = upd\, (p \oplus p', x) \\
&lkp\, (\lambda s.\, upd\, (p\, s, lkp\, (\lambda s'.\, x\, s\, s'))) = lkp\, (\lambda s.\, upd\, (p\, s, x\, s\, (s \downarrow p\, s)))
\end{aligned}
$$

The intuition behind the design of the equation system is that every algebra expression should be rewritable into the form $\mathit{lkp}\,(\lambda s.\ \mathit{upd}\,(p\,s, x\,s))$. Seen as rewrite rules, the 1st equation enables one to prefix a given algebra expression with a pair of occurrences of *lkp* and *upd* whereas the 2nd and 3rd equations allow removal of all subsequent occurrences of *lkp* and *upd*.

The operations

$$\mathit{act} : (S \to P \times X) \to X$$

and

$$\begin{aligned}&\mathit{lkp} : (S \to X) \to X\\ &\mathit{upd} : P \times X \to X\end{aligned}$$

satisfying their respective axioms are interdefinable via

$$\begin{aligned}&\mathit{lkp}\,(\lambda s.\ x\,s) = \mathit{act}\,(\lambda s.\ (\mathsf{o}, x\,s))\\ &\mathit{upd}\,(p, x) = \mathit{act}\,(\lambda s.\ (p, x))\end{aligned}$$

and

$$\mathit{act}\,(\lambda s.\ (p\,s, x\,s)) = \mathit{lkp}\,(\lambda s.\ \mathit{upd}\,(p\,s, x\,s))$$

2.4 Update monads as a compatible composition of reader and writer monads

While state monads cannot be described as compositions of reader and writer monads, update monads are exactly that!

The update monad (T, η, μ) for $(S, (P, \mathsf{o}, \oplus), \downarrow)$ is a compatible composition (in the sense of the definition given in Section A.3) of the reader monad (T_0, η_0, μ_0) for S and the writer monad (T_1, η_1, μ_1) for $(P, \mathsf{o}, \oplus)$: the underlying functor T is the functor composition $T_0 \cdot T_1$ and the unit η and multiplication μ relate to those of the reader and writer monad in a certain way, which implies, in particular, that $T_0 \cdot \eta_1$ is a monad map from (T_0, η_0, μ_0) to (T, η, μ) and $\eta_0 \cdot T_1$ is one from (T_1, η_1, μ_1) to (T, η, μ).

The corresponding distributive law θ of (T_1, η_1, μ_1) over (T_0, η_0, μ_0) is determined by the action $\downarrow$:

$$\begin{aligned}&\theta : \forall\{X\}.\ P \times (S \to X) \to S \to P \times X\\ &\theta\,(p, f) = \lambda s.\ (p, f\,(s \downarrow p))\end{aligned} \tag{1}$$

Moreover, every compatible composition of these two monads is an update monad, since every distributive law θ of (T_1, η_1, μ_1) over (T_0, η_0, μ_0) defines an action $\downarrow$ satisfying (1) via

$$\begin{aligned}&\downarrow : S \times P \to S\\ &s \downarrow p = \mathsf{snd}\,(\theta\,\{S\}\,(p, \mathsf{id}\,\{S\})\,s)\end{aligned} \tag{2}$$

while it follows directly from the definition of θ that

$$p = \mathsf{fst}\,(\theta\,\{S\}\,(p, \mathsf{id}\,\{S\})\,s) \tag{3}$$

Substituting the two definitions into each other the other way around yields identity too, so (1) and (2) give a bijective correspondence between the actions and the distributive laws.

▶ **Lemma 8.** *For any distributive law θ of the writer monad for $(P, \mathsf{o}, \oplus)$ over the reader monad for S, equation (3) holds.*

▶ **Theorem 9.** *Equations (1), (2) establish a bijective correspondence between the actions of* $(P, \mathsf{o}, \oplus)$ *on* S *and the distributive laws of the writer monad for* $(P, \mathsf{o}, \oplus)$ *over the reader monad for* S*.*

For proofs, see Appendix B.

The trivial action $s \downarrow p = s$ corresponds to the distributive law $\theta\,(p, f) = \lambda s.\,(p, f\,s)$, which is the strength of T_0.

As an instance of the general characterization of algebras of a compatible composition of two monads in terms of their algebras, we learn that an algebra of the update monad for $(S, (P, \mathsf{o}, \oplus), \downarrow)$ can be specified as a set X carrying algebras of both the reader and writer monad, i.e., operations

$$\mathit{lkp} : (S \to X) \to X \qquad\qquad \mathit{upd} : P \times X \to X$$

satisfying the conditions

$$x = \mathit{lkp}\,(\lambda s.\,x) \qquad\qquad x = \mathit{upd}\,(\mathsf{o}, x)$$
$$\mathit{lkp}\,(\lambda s.\,\mathit{lkp}\,(\lambda s'.\,x\,s\,s')) = \mathit{lkp}\,(\lambda s.\,x\,s\,s) \qquad \mathit{upd}\,(p, \mathit{upd}(p', x)) = \mathit{upd}\,(p \oplus p', x)$$

plus an additional compatibility condition

$$\mathit{upd}\,(p, \mathit{lkp}\,(\lambda s'.\,x\,s')) = \mathit{lkp}\,(\lambda s.\,\mathit{upd}\,(p, x\,(s \downarrow p)))$$

This axiomatization of *lkp* and *upd* is quite different from the one we showed above—only one axiom is shared—, but nonetheless equivalent. One could also argue that it is more systematic and symmetric.

For the trivial action $s \downarrow p = s$, the compatibility condition becomes $\mathit{upd}\,(p, \mathit{lkp}\,(\lambda s'.\,x\,s'))$ $= \mathit{lkp}\,(\lambda s.\,\mathit{upd}\,(p, x\,s))$—the condition of models of the tensor of the Lawvere theories for reading and writing [9, Section 5].

It is important to notice that algebras (X, upd) of the reader monad are nothing but sets with a *left* action of $(P, \mathsf{o}, \oplus)$ while $(S, \downarrow)$ is a set with a *right* action of $(P, \mathsf{o}, \oplus)$.

2.5 Maps between update monads

What are maps between update monads like? It turns out that, for a suitable notion of act maps, every map between two given acts in the reverse direction defines a map between the corresponding update monads, but this mapping of act maps to monad maps is generally neither injective nor surjective.

We choose to define a *map* between two acts $(S', (P', \mathsf{o}', \oplus'), \downarrow')$ and $(S, (P, \mathsf{o}, \oplus), \downarrow)$ to be a function $t : S' \to S$ together with a monoid homomorphism $q : (P, \mathsf{o}, \oplus) \to (P', \mathsf{o}', \oplus')$ (notice the direction of q!) such that

$$t\,(s \downarrow' q\,p) = t\,s \downarrow p$$

holds.[5]

These pairs (t, q) are in a bijective correspondence with pairs (τ_0, τ_1) where τ_0 is a map between the reader monads (T_0, η_0, μ_0) and (T_0', η_0', μ_0') for S_0 resp. S_0' and τ_1 is a map

[5] More customarily, a map between these acts would be taken to be a function $t : S' \to S$ together with a monoid homomorphism $q : (P', \mathsf{o}', \oplus') \to (P, \mathsf{o}, \oplus)$ satisfying the condition $t\,(s \downarrow' p) = t\,s \downarrow q\,p$, see, e.g., [10, p. 54].

between the writer monads (T_1, η_1, μ_1) and (T_1', η_1', μ_1') for $(P, \mathsf{o}, \oplus)$ resp. $(P, \mathsf{o}', \oplus')$ satisfying the condition

$$\begin{array}{ccc} T_1 \cdot T_0 & \xrightarrow{\tau_1 \cdot \tau_0} & T_1' \cdot T_0' \\ {\scriptstyle\theta}\downarrow & & \downarrow{\scriptstyle\theta'} \\ T_0 \cdot T_1 & \xrightarrow[\tau_0 \cdot \tau_1]{} & T_0' \cdot T_1' \end{array}$$

where θ and θ' are the distributive laws defined by $\downarrow$ resp. $\downarrow'$.

Acts and act maps form a category.

Every act map (t, q) between $(S', (P', \mathsf{o}', \oplus'), \downarrow')$ and $(S, (P, \mathsf{o}, \oplus), \downarrow)$ determines a monad map τ between the corresponding update monads (T, η, μ) and (T, η', μ') via

$$\begin{array}{l} \tau : \forall \{X\}.\, (S \to P \times X) \to S' \to P' \times X \\ \tau\, f = \lambda s.\, \mathsf{let}\ (p, x) = f\, (t\, s)\ \mathsf{in}\ (q\, p, x) \end{array}$$

extending the mapping of acts to monads into a functor from the opposite of the category of acts to the category of monads on **Set**.

This functor is neither faithful nor full. To see the failure of faithfulness, let S be any set, but $S' = 0$, and let $(P, \mathsf{o}, \oplus)$, $(P', \mathsf{o}', \oplus')$ be arbitrary monoids with more than one monoid map between them. Let $\downarrow$ be arbitrary; as $\downarrow' : S' \times P' \to S'$ we can only choose the empty function. Now there is exactly one map $t : S' \to S$, namely the empty function. For any monoid map $q : (P, \mathsf{o}, \oplus) \to (P', \mathsf{o}', \oplus')$, the pair (t, q) is an act map, but the corresponding map τ between the update monads, with type $\tau : \forall \{X\}.\, (S \to P \times X) \to S' \to P' \times X$, sends any given map f to the empty map irrespective of the choice of q.

A simple counterexample to fullness is obtained by considering the reader monad for a given S, the update monad extension of the state monad for S (the update monad of Example 2) and the more interesting one of the two canonical embeddings between them. Concretely, we take S to be an arbitrary non-trivial set (i.e., not 0, not 1) and $S' = S$. We let $(P, \mathsf{o}, \oplus)$ and $\downarrow$ be trivial (i.e., $P = 1$) and we let $(P', \mathsf{o}', \oplus')$ and $\downarrow'$ be the free monoid on the overwrite semigroup on S and the action of P' on S given by overwriting. We define $\tau : \forall \{X\}.\, (S \to 1 \times X) \to S \to (1 + S) \times X$ by $\tau\, f = \lambda s.\, \mathsf{let}\ (*, x) = f\, s\ \mathsf{in}\ (\mathsf{inr}\, s, x)$. Now τ is a monad map, but it is not the image of any act map (t, q).

3 A dependently typed generalization

Recall the fixed-size no-removal buffer and stack of Examples 6 and 7. They can overflow and underflow. This raises a natural question: Is it possible to restrict the updates so that this is guaranteed to not happen?

This cannot be done with the definition given in Section 2.2. The reason is the non-dependence of updates on states. To remedy this, we now define a dependently-typed generalization of update monads. It is related to Abbott, Altenkirch and Ghani's containers [1] (equivalent to (simple, or non-dependent) polynomials [8]).

We recall that a *container* is a set S together with a S-indexed family P. A *map* between two containers (S, P) and (S', P') is given by functions $t : S \to S'$ and $q : \Pi\, \{s : S\}.\, P'\, (t\, s) \to P\ s$. Containers and container morphisms form a monoidal category **Cont**.

Any container determines a set functor $[\![S, P]\!]^{\mathrm{c}}$ (its *interpretation*) by

$$\begin{array}{l} [\![S, P]\!]^{\mathrm{c}}\, X = \Sigma s : S.\, P\, s \to X \\ [\![S, P]\!]^{\mathrm{c}}\, h\, (s, v) = (s, h \circ v) \end{array}$$

By associating a map (t, q) between containers (S, P) and (S', P') with a natural transformation $[\![t, q]\!]^{\mathrm{c}}$ between the functors $[\![S, P]\!]^{\mathrm{c}}$ and $[\![S', P']\!]^{\mathrm{c}}$ by

$$[\![t, q]\!]^{\mathrm{c}}\,(s, v) = (t\,s, v \circ q\,\{s\})$$

the mapping $[\![-]\!]^{\mathrm{c}}$ is extended into a fully faithful monoidal functor from **Cont** to [**Set**, **Set**].

In our previous work with Chapman [2], we introduced directed containers as characterization of containers that are comonads. A directed container is similar to an act, except that the monoid carrier and operations depend on elements of the set.

A *directed container* is a set S together with a S-indexed family P and operations

$$\downarrow : \Pi s : S.\, P\,s \to S$$
$$\mathsf{o} : \Pi\{s : S\}.\, P\,s$$
$$\oplus : \Pi\{s : S\}.\, \Pi p : P\,s.\, P\,(s \downarrow p) \to P\,s$$

satisfying the equations

$$s \downarrow \mathsf{o} = s$$
$$s \downarrow (p \oplus p') = (s \downarrow p) \downarrow p'$$

$$p \oplus \mathsf{o} = p$$
$$\mathsf{o} \oplus p = p$$
$$(p \oplus p') \oplus p'' = p \oplus (p' \oplus p'')$$

Observe that, on the level of terms (with implicit arguments suppressed) these five equations are exactly those of a monoid and an action. But the typing is different. In fact, the 4th equation is only well-typed on the assumption of the 1st equation and similarly the well-typedness of the 5th equation depends on a proof of the 2nd equation. (In the renderings above, this is invisible, as we have also suppressed type-index conversions.) If none of $P\,s$, $\mathsf{o}\,\{s\}$ and $p \oplus \{s\}\,p'$ actually depends on s, the directed container is an act. If $s \downarrow p = s$, then it is a set together with a family of monoids.

A *map* between directed containers $(S, P, \downarrow, \mathsf{o}, \oplus)$ and $(S', P', \downarrow', \mathsf{o}', \oplus')$ is a map (t, q) between the underlying containers (S, P) and (S', P') such that

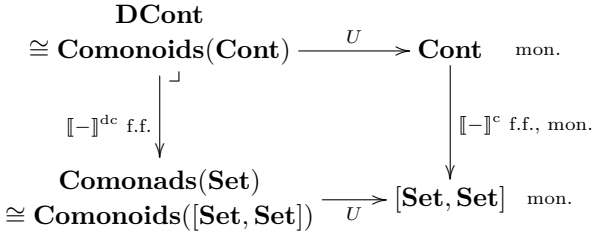
$$t\,(s \downarrow q\,p) = t\,s \downarrow'\, p$$
$$\mathsf{o} = q\,\mathsf{o}'$$
$$q\,p \oplus q\,p' = q\,(p \oplus'\, p')$$

These equation look like those for a monoid map and an action map, but are typed finer. In particular, the 3rd equation is well-typed on the assumption of the 1st equation. If the two directed containers are in fact acts and $q\,\{s\}\,p$ does not actually depend on s, then q is an act map.

Directed containers and directed container maps form a category **DCont** that turns out to be isomorphic to the category **Comonoids**(**Cont**) of comonoid objects in the monoidal category **Cont**.

This isomorphism together with monoidality of the functor $[\![-]\!]^{\mathrm{c}}$ implies that the functor $[\![-]\!]^{\mathrm{c}} : \mathbf{Cont} \to [\mathbf{Set}, \mathbf{Set}]$ lifts to a functor $[\![-]\!]^{\mathrm{dc}}$ from **DCont** $\cong$ **Comonoids**(**Cont**) to **Comonads**(**Set**) $\cong$ **Comonoids**([**Set**, **Set**]) interpreting directed containers into comonads. From fully faithfulness of $[\![-]\!]^{\mathrm{c}}$ it follows that $[\![-]\!]^{\mathrm{dc}}$ is fully faithful too, i.e., the maps between the interpretations of two directed containers are in a bijection between the maps between these directed containers. More, $[\![-]\!]^{\mathrm{dc}}$ is the pullback of $[\![-]\!]^{\mathrm{c}}$ along the forgetful functor $U : \mathbf{Comonads}(\mathbf{Set}) \to [\mathbf{Set}, \mathbf{Set}]$, meaning that directed containers are in fact exactly the

containers whose interpretation carries a comonad structure. This is summarized in the following diagram.



For this paper, we have a reason to refocus from interpretation to "cointerpretation". Let us assign to every container (S, P) a set functor $\langle\!\langle S, P\rangle\!\rangle^{\mathrm{c}}$ (its *cointerpretation*) by setting

$$\langle\!\langle S, P\rangle\!\rangle^{\mathrm{c}}\, X = \Pi s : S.\, P\, s \times X$$
$$\langle\!\langle S, P\rangle\!\rangle^{\mathrm{c}}\, h\, f = \lambda s.\, \mathsf{let}\ (p, x) = f\, s\ \mathsf{in}\ (p, h\, x)$$

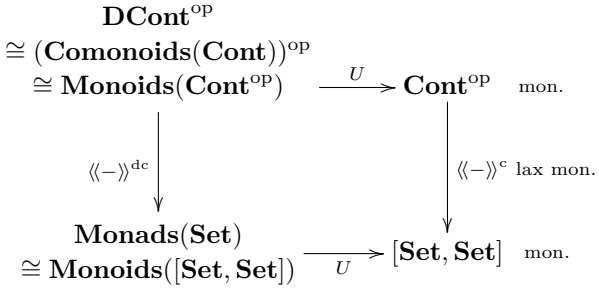
By associating with any container map (t, q) between (S', P') and (S, P) a natural transformation $\langle\!\langle t, q\rangle\!\rangle^{\mathrm{c}}$ between $\langle\!\langle S, P\rangle\!\rangle^{\mathrm{c}}$ and $\langle\!\langle S', P'\rangle\!\rangle^{\mathrm{c}}$, which is easily done by taking

$$\langle\!\langle t, q\rangle\!\rangle^{\mathrm{c}}\, f = \lambda s.\, \mathsf{let}\ (p, x) = f\, (t\, s)\ \mathsf{in}\ (q\, \{s\}\, p, x)$$

we extend the mapping $\langle\!\langle -\rangle\!\rangle^{\mathrm{c}}$ to a functor $\langle\!\langle -\rangle\!\rangle^{\mathrm{c}}$ between $\mathbf{Cont}^{\mathrm{op}}$ and $[\mathbf{Set}, \mathbf{Set}]$.

The functor $\langle\!\langle -\rangle\!\rangle^{\mathrm{c}} : \mathbf{Cont}^{\mathrm{op}} \to [\mathbf{Set}, \mathbf{Set}]$ is not as well-behaved as $[\![-]\!]^{\mathrm{c}} : \mathbf{Cont} \to [\mathbf{Set}, \mathbf{Set}]$. First of all, $\langle\!\langle -\rangle\!\rangle^{\mathrm{c}}$ fails to be monoidal, it is only lax monoidal. Second, it is neither faithful nor full.

Nonetheless, the mere lax monoidality of $\langle\!\langle -\rangle\!\rangle^{\mathrm{c}}$ is enough to obtain a canonical cointerpretation mapping of directed containers into monads. It suffices to note that $\mathbf{DCont}^{\mathrm{op}} \cong (\mathbf{Comonoids}(\mathbf{Cont}))^{\mathrm{op}} \cong \mathbf{Monoids}(\mathbf{Cont}^{\mathrm{op}})$ and $\mathbf{Monads}(\mathbf{Set}) \cong \mathbf{Monoids}([\mathbf{Set}, \mathbf{Set}])$. Lax monoidality of $\langle\!\langle -\rangle\!\rangle^{\mathrm{c}} : \mathbf{Cont}^{\mathrm{op}} \to [\mathbf{Set}, \mathbf{Set}]$ implies that $\langle\!\langle -\rangle\!\rangle^{\mathrm{c}}$ sends monoids to monoids and lifts to a functor $\langle\!\langle -\rangle\!\rangle^{\mathrm{dc}} : \mathbf{DCont}^{\mathrm{op}} \to \mathbf{Monads}(\mathbf{Set})$. This is summarized in the following diagram where the square commutes, but is not a pullback.



Explicitly, the functor $\langle\!\langle -\rangle\!\rangle^{\mathrm{dc}}$ sends a directed container $(S, P, \downarrow, \mathsf{o}, \oplus)$ to the monad (T, η, μ) (the corresponding *(dependently typed) update monad*) given by

$$T\, X = \langle\!\langle S, P\rangle\!\rangle^{\mathrm{c}}\, X = \Pi s : S.\, P\, s \times X$$

$$\eta : \forall \{X\}.\, X \to \Pi s : S.\, P\, s \times X$$
$$\eta\, x = \lambda s.\, (\mathsf{o}, x)$$

$$\mu : \forall \{X\}.\, (\Pi s : S.\, P\, s \times (\Pi s' : S.P\, s' \times X)) \to \Pi s : S.\, P\, s \times X$$
$$\begin{aligned}\mu\, f = \lambda s.\, &\mathsf{let}\ (p, g) = f\, s;\\ &\quad (p', x) = g\, (s \downarrow p)\\ &\mathsf{in}\ (p \oplus p', x)\end{aligned}$$

On the level of terms, the definitions of the unit and multiplication look exactly as those we gave in Section 2.2 for the update monad for an act, but their types are finer.

A map (t, q) between directed containers $(S', P', \downarrow', \mathsf{o}', \oplus')$ and $(S, P, \downarrow, \mathsf{o}, \oplus)$ is sent by $\langle\!\langle - \rangle\!\rangle^{\mathrm{dc}}$ to the natural transformation $\langle\!\langle t, q \rangle\!\rangle^{\mathrm{dc}} = \langle\!\langle t, q \rangle\!\rangle^{\mathrm{c}}$ between the functors $\langle\!\langle S, P \rangle\!\rangle^{\mathrm{c}}$ and $\langle\!\langle S', P' \rangle\!\rangle^{\mathrm{c}}$, which is also a monad map between $\langle\!\langle S, P, \downarrow, \mathsf{o}, \oplus \rangle\!\rangle^{\mathrm{dc}}$ and $\langle\!\langle S', P', \downarrow', \mathsf{o}', \oplus' \rangle\!\rangle^{\mathrm{dc}}$. This way of specifying maps between dependently typed update monads generalizes the one we described in Section 2.5 for maps between simply typed update monads.

The intuitive advantage of dependently typed update monads over simply typed update monads lies in the idea of updates *enabled* (or safe) in a state. In the simply typed case, any update has to apply to any state.

In the dependently typed setting, any initial state $s : S$ determines its own set of updates $P\,s$ enabled in it. And, according to the type of $\downarrow$, only those updates are applicable to s. This means that we are not forced to invent outcomes for updates that should morally only be allowed in some states.

▶ **Example 10.** In the example of the buffer (Example 6), we chose to write only a prefix of a given list into the buffer, if it had no space left for the full list. This is clearly a dangerous design, as values get discarded silently. Another option would have been to introduce a special error state. But with a dependently typed update monad, we can do much better.

The non-overflowing fixed-size no-removal buffer monad is given by the following data:

$$T\,X = \Pi es : E^{\leq N}.\, E^{\leq N - \mathsf{len}\, es} \times X$$

$$\eta : \forall \{X\}.\, X \to \Pi es : E^{\leq N}.\, E^{\leq N - \mathsf{len}\, es} \times X$$
$$\eta\, x = \lambda es.\, []$$

$$\mu : \forall \{X\}.\, (\Pi es : E^{\leq N}.\, E^{\leq N - \mathsf{len}\, es} \times (\Pi es' : E^{\leq N}.\, E^{\leq N - \mathsf{len}\, es'} \times X)) \to \Pi es : E^{\leq N}.\, E^{\leq N - \mathsf{len}\, es} \times X$$
$$\begin{aligned}\mu\, f = \lambda es.\, \mathsf{let}\ & (es', g) = f\, es; \\ & (es'', x) = g\, (es +\!\!+ es') \\ \mathsf{in}\ & (es' +\!\!+ es'', x)\end{aligned}$$

The states are lists of length at most n as before: $S = E^{\leq N}$. But the updates, acceptable lists of values to write, now depend on these states in a natural way. Namely, for a state $es : E^{\leq N}$ of the buffer, the enabled updates are $P\, es = E^{\leq N - \mathsf{len}\, es}$, i.e., lists that can be appended to es without exceeding the length limit N. The means that the action does not have to truncate, it is just concatenation: $es \downarrow es' = es +\!\!+ es'$.

▶ **Example 11.** Similarly, we can amend our two stack monads from Example 7 to be non-underflowing.

As before, the set of states $S = E^*$ is given by lists of elements of E. Regarding the monoid of updates, we can define $P\, es = \{ps : (1 + E)^* \mid \mathsf{removes}\, ps \leq \mathsf{len}\, es\}$ where

$$\begin{aligned}&\mathsf{removes}\, [] = 0 \\ &\mathsf{removes}\, (\mathsf{inl}\, * :: ps) = \mathsf{removes}\, ps + 1 \\ &\mathsf{removes}\, (\mathsf{inr}\, e :: ps) = \mathsf{removes}\, ps \dot{-} 1\end{aligned}$$

Alternatively, we can define $P'\, es = [0..\mathsf{len}\, es] \times E^*$.

Differently from simply typed update monads, dependently typed update monads subsume state monads.

▶ **Example 12.** Given a set S, define $P\,s = S$, $s \downarrow s' = s'$, $\mathsf{o}\,\{s\} = s$, $s' \oplus \{s\}\,s'' = s''$.

The update monad for this directed container is the state monad for S.

In Example 2, we noted that the state monad for S fails to be a simply typed update monad, because $P = S$ is just a semigroup, not a monoid. With the dependently typed notion, we can afford a different unit element $\mathsf{o}\,\{s\} = s : P\,s = S$ for each $s : S$, overcoming this obstacle.

The monad morphisms *idem*, *sec* and *retr* are morphisms of dependently typed update monads.

An (EM-)algebra for the dependently typed update monad for the directed container $(S, P, \downarrow, \mathsf{o}, \oplus)$ is a set X with an operation

$$act : (\Pi s : S.\, P\,s \times X) \to X$$

satisfying the equations

$$x = act\,(\lambda s.\,(\mathsf{o}\,\{s\}, x))$$
$$act\,(\lambda s.\,(p\,s, act\,(\lambda s'.\,(p'\,s\,s', x\,s\,s')))) = act\,(\lambda s.\,(p\,s \oplus \{s\}\,p'\,s\,(s \downarrow p\,s), x\,s\,(s \downarrow p\,s)))$$

Again the equations look exactly the same as in the simply typed case, but the types are different.

It is not clear to us whether dependently typed update monads admit a useful decomposition similar to the decomposition of simple update monads into reader and writer monads. One possibility is to resort to relative monads of Altenkirch et al. [4]: dependently typed update monads can be described as compatible compositions of certain relative monads.

Given a directed container $(S, P, \downarrow, \mathsf{o}, \oplus)$. Define $J_0 : [S, \mathbf{Set}] \to \mathbf{Set}$ by $J_0\,X = \Pi s : S.\, X\,s$ and $J_1 : \mathbf{Set} \to [S, \mathbf{Set}]$ by $J_1\,X\,s = X$ (notice that J_0 is right adjoint to J_1). Now on J_0 we have a rather trivial but nonetheless reader-like relative monad $(T_0, \eta_0, (-)_0^*)$ given by $T_0\,X = \Pi s : S.\, X\,s = J_0\,X$, $\eta_0\,\{X\} = \mathsf{id}\,\{J_0\,X\}$, $k_0^* = k$. On J_1 at the same time we can define a writer-like relative monad $(T_1, \eta_1, (-)_1^*)$ by $T_1\,X\,s = P\,s \times X$, $\eta_1\,\{X\}\,\{s\}\,x = (\mathsf{o}\,\{s\}, x)$, $k_1^*\,\{s\}\,(p, x) = \mathsf{let}\ (p', y) = k\,\{s \downarrow p\}\,x\ \mathsf{in}\ (p \oplus \{s\}\,p', y)$. The dependently typed update monad is a compatible composition of the two relative monads.

4 Kammar and Plotkin's generalization of state monads

Kammar and Plotkin[6] have proposed a generalization of state monads that is related to ours. Similarly to us, they employ monoids and monoid actions. Kammar and Plotkin's monad for an act $(S, (P, \mathsf{o}, \oplus), \downarrow)$ is defined by

$$T\,X = \Pi s : S.\,(s \downarrow P) \times X$$

$$\eta : \forall\{X\}.\, X \to \Pi s : S.\,(s \downarrow P) \times X$$
$$\eta\,x = \lambda s.\,(s, x)$$

$$\mu : \forall\{X\}.\,(\Pi s : S.\,(s \downarrow P) \times (\Pi s' : S.\,(s' \downarrow P) \times X)) \to \Pi s : S.\,(s \downarrow P) \times X$$
$$\mu\,f = \lambda s.\,\mathsf{let}\ (s', g) = f\,s;$$
$$\qquad\qquad (s'', x) = g\,s'$$
$$\qquad \mathsf{in}\ (s'', x)$$

[6] O. Kammar and G. Plotkin. Take action for your state: effective conservative restrictions. Slides from Scottish Programming Language Seminar, Strathclyde, Nov. 2010.

Here $s \downarrow P = \{s \downarrow p \mid p \in P\} \subseteq S$ is the orbit of s along the action $\downarrow$ of the monoid $(P, \mathsf{o}, \oplus)$ on the set S. Notice that η and μ are defined just as for the state monad for S, only the typing is finer. In particular, the monoid structure and the action only appear in the types.

Reader and state monads are special cases of this unification, while writer monads are not. (Remember that, in contrast, simply typed update monads cover reader and writer monads, but not state monads.)

Kammar and Plotkin's monad for $(S, (P, \mathsf{o}, \oplus), \downarrow)$ turns out to be the middle monad in the epi-mono factorization of the obvious monad map τ between the simply typed update monad for $(S, (P, \mathsf{o}, \oplus), \downarrow)$ and the state monad for S. For a given state, τ just applies to a given initial state the update that it produces.

$$\tau : \forall\{X\}.\, (S \to P \times X) \to S \to S \times X$$
$$\tau\, f = \lambda s.\, \mathsf{let}\ (p, x) = f\, s\ \mathsf{in}\ (s \downarrow p, x)$$

Just as the state monad, Kammar and Plotkin's monad is an instance of a dependently typed update monad. The appropriate directed container is $(S, P', \downarrow', \mathsf{o}', \oplus')$ where

$$P'\, s = s \downarrow P$$

$$\downarrow' : \Pi s : S.\, \Pi s' : s \downarrow P.\, S$$
$$s \downarrow' s' = s'$$

$$\mathsf{o}' : \Pi\{s : S\}.\, s \downarrow P$$
$$\mathsf{o}'\, \{s\} = s$$

$$\oplus' \colon \Pi s : S.\, \Pi s' : s \downarrow P.\, s' \downarrow P \to s \downarrow P$$
$$s' \oplus'\, \{s\}\, s'' = s''$$

5 Conclusion and future work

We have presented some facts about a class of monads that we call update monads. We hope that those convince the reader that the concept is meaningful and elegant. Although we arrived at update monads thinking about cointerpretation of directed containers, in hindsight we think that they are above all a simple, but instructive unification of the reader, writer and state monads. This unification helps explain how they and some further special monads interrelate and why.

As future work, we wish to generalize this work to monoidal closed categories (replacing unique comonoids with arbitrary comonoids) and to presheaf categories (replacing directed containers with directed indexed containers).

Acknowledgements. Danel Ahman thanks Ohad Kammar for discussions. Tarmo Uustalu acknowledges the feedback from Ichiro Hasuo, Zhenjiang Hu and their colleagues at the University of Tokyo and National Institute of Informatics.

References

1 M. Abbott, T. Altenkirch, N. Ghani. Containers: constructing strictly positive types. *Theor. Comput. Sci.*, v. 342, n. 1, pp. 3–27, 2005.

2 D. Ahman, J. Chapman, T. Uustalu. When is a container a comonad? In L. Birkedal, ed., *Proc. of 15th Int. Conf. on Foundations of Software Science and Computation Structures, FoSSaCS 2012 (Tallinn, March 2012)*, v. 7213 of *Lect. Notes in Comput. Sci.*, pp. 74–88. Springer, 2012. Journal version to appear in *Log. Methods in Comput. Sci.*

3 D. Ahman, T. Uustalu. Distributive laws of directed containers. *Progress in Informatics*, v. 10, pp. 3–18, 2013.

4 T. Altenkirch, J. Chapman, T. Uustalu. Monads need not be endofunctors. In L. Ong, ed., *Proc. of 13th Int. Conf. on Foundations of Software Science and Computation Structures, FoSSaCS 2010 (Paphos, March 2010)*, v. 6014 of *Lect. Notes in Comput. Sci.*, pp. 297-311. Springer, 2010. Journal version to appear in *Log. Methods in Comput. Sci.*

5 M. G. Brin. On the Zappa-Szép product. *Commun. in Algebra*, v. 33, n. 2, pp. 393–424, 2005.

6 M. Barr and C. Wells. *Toposes, Triples and Theories*, v. 278 of *Grundlehren der mathematischen Wissenschaften*. Springer, 1984.

7 J. Beck. Distributive laws. In B. Eckmann, ed., *Seminar on Triples and Categorical Homology, ETH 1966/67*, v. 80 of *Lect. Notes in Math.*, pp. 119–140. Springer, 1969.

8 N. Gambino, M. Hyland. Wellfounded trees and dependent polynomial functors. In S. Berardi, M. Coppo, F. Damiani, eds., *Revised Selected Papers from Int. Wksh. on Types for Proofs and Programs, TYPES 2003 (Torino, Apr./May 2003)*, v. 2085 of *Lect. Notes in Comput. Sci.*, pp. 210–225. Springer, 2004.

9 M. Hyland, G. Plotkin, J. Power. Combining effects: sum and tensor. *Theor. Comput. Sci.*, v. 357, no. 1, pp. 70–99, 2006.

10 M. Kilp, U. Knauer, A. V. Mikhalev. *Monoids, Acts and Categories: With Applications to Wreath Products and Graphs*, v. 29 of *De Gruyter Expositions in Mathematics*. De Gruyter, 2000.

11 S. Mac Lane. *Categories for the Working Mathematician*, v. 5 of *Graduate Texts in Mathematics*. Springer, 1971.

12 U. Norell. Towards a practical programming language based on dependent type theory. PhD thesis, Chalmers University of Technology, 2007.

13 M. Piróg, J. Gibbons. Monads for behavior. In D. Kozen, M. Mislove, eds., *Proc. of MFPS XXIX (New Orleans, LA, June 2013)*, v. 298 of *Electron. Notes in Theor. Comput. Sci.*, pp. 309–324. Elsevier, 2013.

14 G. D. Plotkin, J. Power. Notions of computation determine monads. In M. Nielsen, U. Engberg, eds., *Proc. of 5th Int. Conf. on Foundations of Software Science and Computation Structures, FoSSaCS 2002 (Grenoble, April 2002)*, v. 2303 of *Lect. Notes in Comput. Sci.*, pp. 342–356. Springer, 2002.

15 P. Wadler. The essence of functional programming. In *Proc. of 19th Ann. ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages, POPL 1992 (Albuquerque, NM, Jan. 1992)*, pp. 1–14. ACM Press, 1992.

A Background

A.1 Monoids, actions

We recall that a *monoid* is a set P together with two operations

$$\mathsf{o} : P$$
$$\oplus : P \times P \to P$$

satisfying

$$p \oplus \mathsf{o} = p$$
$$\mathsf{o} \oplus p = p$$
$$(p \oplus p') \oplus p'' = p \oplus (p' \oplus p'')$$

A *map* between two monoids $(P, \mathsf{o}, \oplus)$ and $(P', \mathsf{o}', \oplus')$ is a map

$$q : P \to P'$$

satisfying

$$q\,\mathsf{o} = \mathsf{o}'$$
$$q\,(p \oplus p') = q\,p \oplus' q\,p'$$

A *right action* of a monoid $(P, \mathsf{o}, \oplus)$ on a set S is an operation

$$\downarrow : S \times P \to S$$

satisfying the conditions

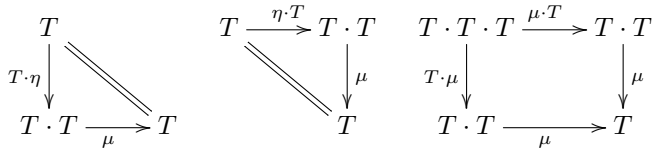
$$s \downarrow \mathsf{o} = s$$
$$s \downarrow (p \oplus p') = (s \downarrow p) \downarrow p'$$

Similarly, a *left action* of $(P, \mathsf{o}, \oplus)$ on S is an operation $\uparrow : P \times S \to S$ satisfying $\mathsf{o} \uparrow s = s$, $(p \oplus p') \uparrow s = p \uparrow (p' \uparrow s)$.

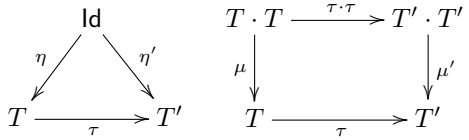
A.2 Monads, monad algebras

We recall the definitions of monads and algebras for monads. For thorough expositions, we refer the reader to the books by Barr and Wells [6, Ch. 3] and Mac Lane [11, § VI].

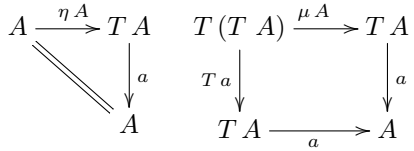
A *monad* on a category $\mathbb{C}$ is given by an endofunctor T on $\mathbb{C}$ and natural transformations $\eta : \mathsf{Id} \to T$ and $\mu : T \cdot T$ satisfying the conditions



A *map* between monads (T, η, μ) and (T', η', μ') on the same category $\mathbb{C}$ is a natural transformation $\tau : T \to T'$ satisfying the conditions



An *(Eilenberg-Moore) algebra* of a monad (T, η, μ) is an object A together with a map $a : T\,A \to A$ satisfying the conditions



For any object A, there is a free algebra of the monad (T, η, μ) on A: the algebra $(T\,A, \mu\,A)$ together with the map $\eta\,A : A \to T\,A$.

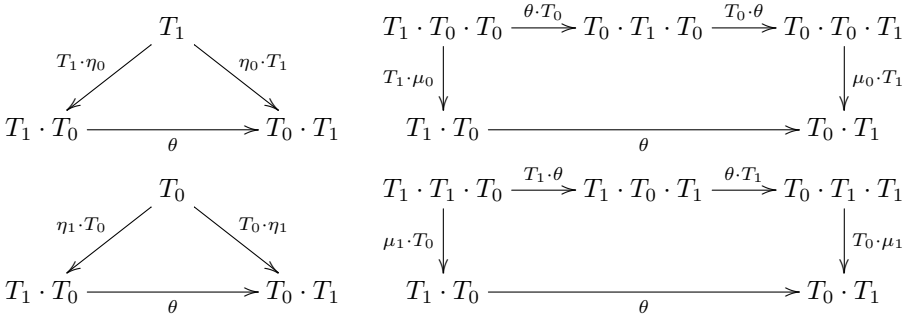
A.3 Distributive laws and compatible compositions of monads

Distributive laws, compatible compositions and liftings are due to Beck [7]. They are also discussed in the book of Barr and Wells [6, Ch. 9].

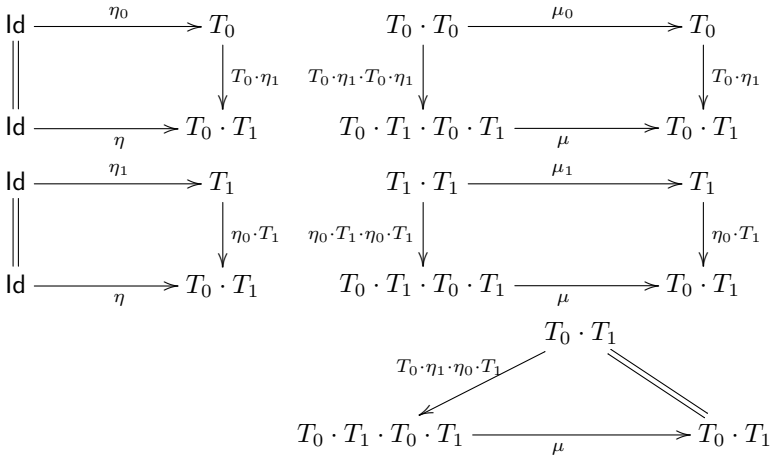
A *distributive law* between two monads (T_0, η_0, μ_0) and (T_1, η_1, μ_1) on the same category $\mathbb{C}$ is a natural transformation

$$\theta : T_1 \cdot T_0 \to T_0 \cdot T_1$$

satisfying the conditions



A *compatible composition* of monads (T_0, η_0, μ_0) and (T_1, η_1, μ_1) on $\mathbb{C}$ is a monad structure (η, μ) on the endofunctor $T_0 \cdot T_1$ satisfying the conditions



Notice that the first two conditions say that $T_0 \cdot \eta_1$ is a morphism between the monads (T_0, η_0, μ_0) and $(T_0 \cdot T_1, \eta, \mu)$ and the next two say that $\eta_0 \cdot T_1$ is a morphism between the monads (T_1, η_1, μ_1) and $(T_0 \cdot T_1, \eta, \mu)$. Notice also that the 1st and 3rd conditions really say the same, namely, that $\eta = \eta_0 \cdot \eta_1$. The most significant condition is the 5th, the so-called middle unital law.

Distributive laws and compatible compositions are in a bijective correspondence. A distributive law θ determines a compatible composition (η, μ) via

$$\eta = \mathsf{Id} \xrightarrow{\eta_0 \cdot \eta_1} T_0 \cdot T_1$$
$$\mu = T_0 \cdot T_1 \cdot T_0 \cdot T_1 \xrightarrow{T_0 \cdot \theta \cdot T_1} T_0 \cdot T_0 \cdot T_1 \cdot T_1 \xrightarrow{\mu_0 \cdot \mu_1} T_0 \cdot T_1$$

Conversely, a compatible composition (η, μ) defines a distributive law θ via

$$\theta = T_1 \cdot T_0 \xrightarrow{\eta_0 \cdot T_1 \cdot T_0 \cdot \eta_1} T_0 \cdot T_1 \cdot T_0 \cdot T_1 \xrightarrow{\mu} T_0 \cdot T_1$$

Given a distributive law θ between two monads (T_0, η_0, μ_0) and (T_1, η_1, μ_1), a pair of their algebras (A, a_0) and (A, a_1) with the same carrier is *matching*, if it satisfies the condition

$$\begin{array}{ccccc} T_1\,(T_0\,A) & \xrightarrow{\theta\,A} & T_0\,(T_1\,A) & \xrightarrow{T_0\,a_1} & T_0\,A \\ {\scriptstyle T_1\,a_0}\downarrow & & & & \downarrow{\scriptstyle a_0} \\ T_1\,A & & \xrightarrow[a_1]{} & & A \end{array}$$

Matching pairs of algebras are in a bijective correspondence with algebras of the compatible composition.

A matching pair of algebras (A, a_0, a_1) defines an algebra (A, a) via

$$a = \; T_0\,(T_1\,A) \xrightarrow{T_0\,a_1} T_0\,A \xrightarrow{a_0} A$$

An algebra (A, a) induces a matching pair (A, a_0, a_1) via

$$a_0 = \; T_0\,A \xrightarrow{T_0\,(\eta_1\,A)} T_0\,(T_1\,A) \xrightarrow{a} A$$
$$a_1 = \; T_1\,A \xrightarrow{\eta_0\,(T_1\,A)} T_0\,(T_1\,A) \xrightarrow{a} A$$

The counterpart under this bijection of the free algebra $(T_0\,(T_1\,A), \mu\,A)$ of the compatible composition on A is the matching pair $(T_0\,(T_1\,A), \bar{\mu}_0\,A, \bar{\mu}_1\,A)$ where

$$\bar{\mu}_0 = \; T_0 \cdot T_0 \cdot T_1 \xrightarrow{\mu_0 \cdot T_1} T_0 \cdot T_1$$
$$\bar{\mu}_1 = \; T_1 \cdot T_0 \cdot T_1 \xrightarrow{\theta \cdot T_1} T_0 \cdot T_1 \cdot T_1 \xrightarrow{T_0 \cdot \mu_1} T_0 \cdot T_1$$

B Proof of the main theorem

B.1 Proof of Lemma 8

$$\begin{array}{ll} & p \\ = & \\ & \mathsf{fst}\,((\lambda s'.\,(p, *))\,s) \\ = & \quad \{\text{def. of } \eta_0\} \\ & \mathsf{fst}\,((\eta_0 \cdot T_1)\,\{1\}\,(p, *)\,s) \\ = & \quad \{\text{distr. law eq. 1 for } \theta\} \\ & \mathsf{fst}\,((\theta \circ T_1 \cdot \eta_0)\,\{1\}\,(p, *)\,s) \\ = & \quad \{\text{def. of } \eta_0\} \\ & \mathsf{fst}\,(\theta\,\{1\}\,(p, \lambda s'.\,*)\,s) \\ = & \quad \{\text{defs. of } T_1,\, T_0\} \\ & \mathsf{fst}\,((\theta\,\{1\} \circ (T_1 \cdot T_0)\,(\lambda s'.\,*))\,(p, f)\,s) \\ = & \quad \{\text{naturality of } \theta\} \\ & \mathsf{fst}\,(((T_0 \cdot T_1)\,(\lambda s'.\,*) \circ \theta\,\{X\})\,(p, f)\,s) \\ = & \quad \{\text{defs. of } T_0,\, T_1\} \\ & \mathsf{fst}\,(\theta\,\{X\})\,(p, f)\,s) \end{array}$$

B.2 Proof of Theorem 9

Given an action $\downarrow$, we must verify that $\theta : \forall\{X\}.\,T_1\,(T_0\,X) \to T_0\,(T_1\,X)$ defined by $\theta\,\{X\}\,(p, f) = \lambda s.\,(p, f\,(s \downarrow p))$ is a distributive law.

Proof of naturality of θ.

$$
\begin{array}{ll}
 & ((T_0 \cdot T_1)\, g \circ \theta\, \{X\})\, (p, f) \\
= & \quad \{\text{def. of } \theta\} \\
 & (T_0 \cdot T_1)\, g\, (\lambda s.\, (p, f\, (s \downarrow p))) \\
= & \quad \{\text{def. of } T_0,\, T_1\} \\
 & \lambda s.\, (p, g\, (f\, (s \downarrow p))) \\
= & \quad \{\text{def. of } \theta\} \\
 & \theta\, \{Y\}\, (p, g \circ f) \\
= & \quad \{\text{def. of } T_1,\, T_0\} \\
 & (\theta\, \{Y\} \circ (T_1 \cdot T_0)\, g)\, (p, f)
\end{array}
$$

Proof of distributive law equation 1 for θ.

$$
\begin{array}{ll}
 & (\theta \circ T_1 \cdot \eta_0)\, \{X\}\, (p, x) \\
= & \quad \{\text{defs. of } T_1,\, \eta_0\} \\
 & \theta\, \{X\}\, (p, \lambda s'.\, x) \\
= & \quad \{\text{def. of } \theta\} \\
 & \lambda s.\, (p, (\lambda s'.\, x)\, (s \downarrow p)) \\
= & \\
 & \lambda s.\, (p, x) \\
= & \quad \{\text{def. of } \eta_0\} \\
 & (\eta_0 \cdot T_1)\, \{X\}\, (p, x)
\end{array}
$$

Proof of distributive law equation 2 for θ.

$$
\begin{array}{ll}
 & (\theta \circ T_1 \cdot \mu_0)\, \{X\}\, (p, f) \\
= & \quad \{\text{defs. of } T_1,\, \mu_0\} \\
 & \theta\, \{X\}\, (p, \lambda s'.\, f\, s'\, s') \\
= & \quad \{\text{def. of } \theta\} \\
 & \lambda s.\, (p, f\, (s \downarrow p)\, (s \downarrow p)) \\
= & \quad \{\text{def. of } \mu_0\} \\
 & (\mu_0 \cdot T_1)\, \{X\}\, (\lambda s.\, \lambda s'.\, (p, f\, (s \downarrow p)\, (s' \downarrow p))) \\
= & \quad \{\text{defs. of } T_0,\, \theta\} \\
 & (\mu_0 \cdot T_1 \circ T_0 \cdot \theta)\, \{X\}\, (\lambda s.\, (p, f\, (s \downarrow p))) \\
= & \quad \{\text{def. of } \theta\} \\
 & (\mu_0 \cdot T_1 \circ T_0 \cdot \theta \circ \theta \cdot T_0)\, \{X\}\, (p, f)
\end{array}
$$

Proof of distributive law equation 3 for θ.

$$
\begin{array}{ll}
 & (\theta \circ \eta_1 \cdot T_0)\, \{X\}\, f \\
= & \quad \{\text{def. of } \eta_1\} \\
 & \theta\, \{X\}\, (\mathsf{o}, f) \\
= & \quad \{\text{def. of } \theta\} \\
 & \lambda s.\, (\mathsf{o}, f\, (s \downarrow \mathsf{o})) \\
= & \quad \{\text{action eq. 1 for } \downarrow\} \\
 & \lambda s.\, (\mathsf{o}, f\, s) \\
= & \quad \{\text{defs. of } T_0,\, \eta_1\} \\
 & (T_0 \cdot \eta_1)\, \{X\}\, f
\end{array}
$$

Proof of distributive law equation 4 for θ.

$$
\begin{aligned}
& (\theta \circ \mu_1 \cdot T_0)\,\{X\}\,(p, (p', f)) \\
= \quad & \{\text{def. of } \mu_1\} \\
& \theta\,\{X\}\,(p \oplus p', f) \\
= \quad & \{\text{def. of } \theta\} \\
& \lambda s.\,(p \oplus p', f\,(s \downarrow (p \oplus p'))) \\
= \quad & \{\text{action eq. 2 for } \downarrow\} \\
& \lambda s.\,(p \oplus p', f\,((s \downarrow p) \downarrow p')) \\
= \quad & \{\text{defs. of } T_0,\ \mu_1\} \\
& (T_0 \cdot \mu_1)\,\{X\}\,(\lambda s.\,(p, (p', f\,((s \downarrow p) \downarrow p')))) \\
= \quad & \{\text{def. of } \theta\} \\
& (T_0 \cdot \mu_1 \circ \theta \cdot T_1)\,\{X\}\,(p, \lambda s.\,(p', f\,(s \downarrow p'))) \\
= \quad & \{\text{defs. of } T_1,\ \theta\} \\
& (T_0 \cdot \mu_1 \circ \theta \cdot T_1 \circ T_1 \circ \theta)\,\{X\}\,(p, (p', f))
\end{aligned}
$$

Given a distributive law θ, we must verify that $\downarrow\colon S \times P \to S$ defined by $s \downarrow p = \mathsf{snd}\,(\theta\,\{S\}\,(p, \lambda s'.\,s')\,s)$ is an action.

Proof of action law 1 for $\downarrow$.

$$
\begin{aligned}
& s \downarrow \mathsf{o} \\
= \quad & \{\text{def. of } \downarrow\} \\
& \mathsf{snd}\,(\theta\,\{S\}\,(\mathsf{o}, \lambda s'.\,s')\,s) \\
= \quad & \{\text{def. of } \eta_1\} \\
& \mathsf{snd}\,((\theta \circ \eta_1 \cdot T_0)\,\{S\}\,(\lambda s'.\,s')\,s) \\
= \quad & \{\text{distr. law eq. 3 for } \theta\} \\
& \mathsf{snd}\,((T_0 \cdot \eta_1)\,\{S\}\,(\lambda s'.\,s')\,s) \\
= \quad & \{\text{defs. of } T_0,\ \eta_1\} \\
& \mathsf{snd}\,((\lambda s'.\,(\mathsf{o}, s'))\,s) \\
= & \\
& s
\end{aligned}
$$

Proof of action law 2 for $\downarrow$.

$$
\begin{aligned}
& s \downarrow (p \oplus p') \\
= \quad & \{\text{def. of } \downarrow\} \\
& \mathsf{snd}\,(\theta\,\{S\}\,(p \oplus p', \lambda s'.\,s')\,s) \\
= \quad & \{\text{def. of } \mu_1\} \\
& \mathsf{snd}\,((\theta \circ \mu_1 \cdot T_0)\,\{S\}\,(p, (p', \lambda s'.\,s'))\,s) \\
= \quad & \{\text{distr. law eq. 4 for } \theta\} \\
& \mathsf{snd}\,((T_0 \cdot \mu_1 \circ \theta \cdot T_1 \circ T_1 \cdot \theta)\,\{S\}\,(p, (p', \lambda s'.\,s'))\,s) \\
= \quad & \{\text{def. of } T_1,\ \text{Lemma 8, def. of } \downarrow\} \\
& \mathsf{snd}\,((T_0 \cdot \mu_1 \circ \theta \cdot T_1)\,\{S\}\,(p, \lambda s'.\,(p', s' \downarrow p'))\,s) \\
= \quad & \{\text{defs. of } T_0,\ T_1\} \\
& \mathsf{snd}\,((T_0 \cdot \mu_1 \circ \theta \cdot T_1 \circ (T_1 \cdot T_0)\,(\lambda s'.\,(p', s' \downarrow p')))\,\{S\}\,(p, \lambda s'.\,s'))\,s) \\
= \quad & \{\text{naturality of } \theta\} \\
& \mathsf{snd}\,((T_0 \cdot \mu_1 \circ (T_0 \cdot T_1)\,(\lambda s'.\,(p', s' \downarrow p')) \circ \theta)\,\{S\}\,(p, \lambda s'.\,s'))\,s) \\
= \quad & \{\text{Lemma 8, def. of } \downarrow\} \\
& \mathsf{snd}\,((T_0 \cdot \mu_1 \circ (T_0 \cdot T_1)\,(\lambda s'.\,(p', s' \downarrow p')))\,\{S\}\,(\lambda s'.\,(p, s' \downarrow p))\,s)
\end{aligned}
$$

$$
\begin{array}{ll}
= & \{\text{defs. of } T_0,\, T_1\} \\
& \mathsf{snd}\,((T_0 \cdot \mu_1)\,\{S\}\,(\lambda s'.\,(p, (p', (s' \downarrow p) \downarrow p')))\,s) \\
= & \{\text{defs. of } T_0,\, \mu_1\} \\
& \mathsf{snd}\,((\lambda s'.\,(p \oplus p', (s' \downarrow p) \downarrow p'))\,s) \\
= & \\
& (s \downarrow p) \downarrow p'
\end{array}
$$

Finally, we have to check that the correspondence is bijective.

$$
\begin{array}{ll}
& s \downarrow' p \\
= & \{\text{def. of } \downarrow'\} \\
& \mathsf{snd}\,(\theta\,\{S\}\,(p, \mathsf{id}\,\{S\})\,s) \\
= & \{\text{def. of } \theta\} \\
& \mathsf{snd}\,(p, \mathsf{id}\,\{S\}\,(s \downarrow p)) \\
= & \\
& s \downarrow p
\end{array}
$$

$$
\begin{array}{ll}
& \theta'\,\{X\}\,(p, f) \\
= & \{\text{def. of } \theta'\} \\
& \lambda s.\,(p, f\,(s \downarrow p)) \\
= & \{\text{Lemma 8, def. of } \downarrow\} \\
& \lambda s.\,(\mathsf{fst}\,(\theta\,\{S\}\,(p, \mathsf{id}\,\{S\})\,s), f\,(\mathsf{snd}\,(\theta\,\{S\}\,(p, \mathsf{id}\,\{S\})\,s))) \\
= & \{\text{defs. of } T_0,\, T_1\} \\
& ((T_0 \cdot T_1)\,f \circ \theta\,\{S\})\,(p, \mathsf{id}\,\{S\}) \\
= & \{\text{naturality of } \theta\} \\
& (\theta\,\{X\} \circ (T_1 \cdot T_0)\,f)\,(p, \mathsf{id}\,\{S\}) \\
= & \{\text{defs. of } T_0,\, T_1\} \\
& \theta\,\{X\}\,(p, f)
\end{array}
$$

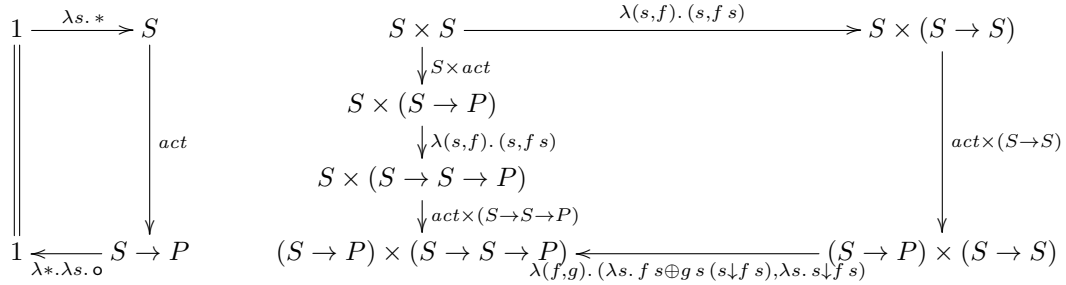
C Algebras of update monads as models of Lawvere theories

In Sections 2.3 and 2.4, we showed three different equivalent definitions of algebras for the update monad for a given act $(S, (P, \mathsf{o}, \oplus), \downarrow)$. An algebra is the same as a model of the (generally large) Lawvere theory corresponding to this monad. Each of the three definitions corresponds to a particular presentation of this Lawvere theory.

The first presentation is given by one operation

$$act : S \to S \to P$$

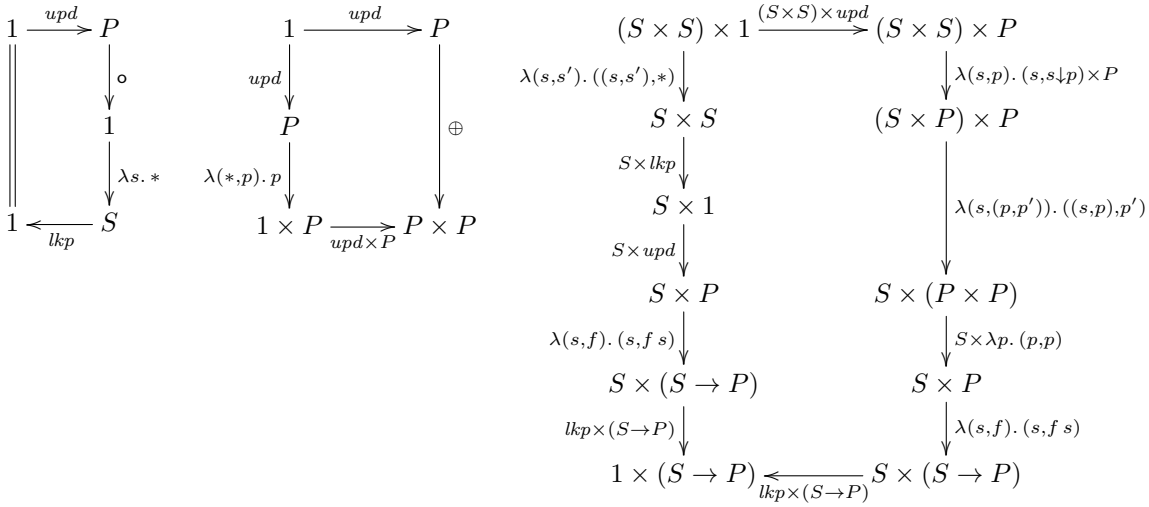
and two equations



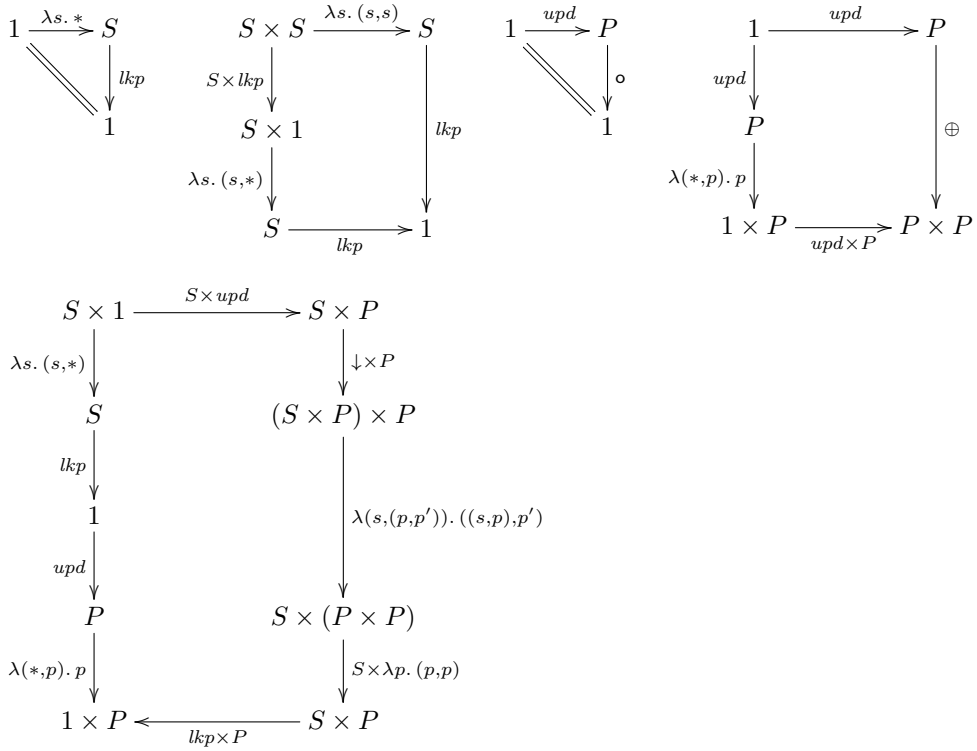
The second and third resemble Melliès's and Plotkin's variations[7] of Plotkin and Power's presentation of the theory of (global) state [14]. The second is given by two operations

$lkp : S \to 1$
$upd : 1 \to P$

satisfying



The third has the same operations, but different equations:



[7] P.-A. Melliès. String diagrams in logic and computer science. Slides from lecture 6 from course held at ITU Copenhagen, Apr. 2011. G. Plotkin. Algebraic effects. Slides from Logic and Interaction, Marseille, Feb. 2012.