An equational modeling of asynchronous concurrent programming

David Janin

Univ. Bordeaux, CNRS, Bordeaux INP,
LaBRI, UMR 5800
France
janin@labri.fr

Abstract. Asynchronous concurrent programing is a widely spread technique offering some simple concurrent primitives that are restricted in such a way that the resulting concurrent programs are dead-lock free. In this paper, we develop and study a formal model of the underlying application programmer interface. For such a purpose, we formally define the extension of a monad by some notion of monad references uniquely bound to running monad actions together with the associated asynchronous primitives fork and read. The expected semantics is specified via two series of equations relating the behavior of these extension primitives with the underlying monad primitives. Thanks to these equations, we recover a fairly general notion of promises and prove that they induce a monad isomorphic to the underlying monad. We also show how synchronous and asynchronous reactive data flow programming eventually derive from such a formalization of asynchronous concurrency, uniformly lifting fork and read primitives from monadic actions to monadic streams of actions. Our proposal is illustrated throughout by concrete extensions of Haskell IO monad that allows for assessing the validity of the proposed equations and for showing how closely our proposal is related with existing libraries.

1 Introduction

Asynchronous programming with promises. Asynchronous programming is quite a popular approach for programming lightly concurrent applications such as, for instance, web services. Based on *promises*, a notion introduced in the late 70s and eventually integrated into concurrent extension of functional programing languages such as Lisp [3] or ML [10], asynchronous concurrent programming is nowadays available in most modern programing languages, including typed languages such as OCaml [8] and Haskell [7].

One of the reason of such a success is that asynchronous programing is both comfortable and safe. Comfort comes from asynchronism, safety comes from deadlock freeness. Indeed, most asynchronous libraries allow for forking programs while keeping promises of their returned values. Provided no other communication mechanisms are used, the dependency graph resulting from creating and reading promises is acyclic therefore deadlock free.

The monadic nature of promises. Quite interestingly, in most libraries, promises are defined with some *flavor* of a monad and discussions about the true monadic nature of promises are numerous on the web, with various and contradictory conclusions depending on the considered host language and library. Indeed, each asynchronous concurrent programming library offers its own interface, mostly incompatible with the others. This does not help having a clear picture of what promises truly are.

In Haskell for instance, asynchronous programing is provided by the *async* library. In this library, there is the type $Async\ a$, which elements may sound like promises of values. They are created by $async :: IO\ a \to IO\ (Async\ a)$, that acts as a fork, and used by $wait :: Async\ a \to IO\ a$, that acts as a read. However, while $Async$ indeed has a functor instance, it does not have any monad instance. Instead, observing the above types, one may try to define promises as elements of type $IO\ (Async\ a)$. A priori, such an idea makes a lot of sense since there is the return function definable by:

$$\begin{aligned}&returnAsync :: a \to IO\ (Async\ a)\\&returnAsync\ a = async\ (return\ a)\end{aligned}$$

and at least four distinct candidates for an associated bind function:

$$bindAsync :: IO\ (Async\ a) \to (a \to IO\ (Async\ b)) \to IO\ (Async\ b)$$

(a) $bindAsync\ m\ f = async\ (m \ggg\!= wait \ggg\!= f \ggg\!= wait)$
(b) $bindAsync\ m\ f = m \ggg\!= \lambda r \to async\ (wait\ r \ggg\!= f \ggg\!= wait)$
(c) $bindAsync\ m\ f = m \ggg\!= wait \ggg\!= \lambda a \to async\ (f\ a \ggg\!= wait)$
(d) $bindAsync\ m\ f = m \ggg\!= wait \ggg\!= f$

for every $m :: IO\ (Async\ a)$ and $f :: a \to IO\ (Async\ b)$. However, none of them induces a valid monad instance. In Haskell, the possibility of a specific monad of promises seems essentially lost in the surrounding IO monad.

On the contrary, in a language like OCaml where there is no IO monad but an implicit one, the monadic flavor of promises is made explicit. In both *lwt* and *async* OCaml libraries, binding the fullfillement of a promise with some callback function is allowed by a *bind* function, and simple promises are created by a *return* function [8]. In other words, OCaml promises are presented as if they form a monad. Moreover, as OCaml is a strict language, this monadic point of view is somehow crucial for dynamically creating and combining simple programs into more complex ones that can later on be forked.

Main goal. Whether or not these libraries actually define valid monad instances seems, at best, to be left unproved. If so, one may also wonder how such valid monads of promises may exist in OCaml while there does not seem to be one in Haskell.

Our goal in this paper is to investigate such an apparent contradiction. For such a purpose, we shall formally investigate the monadic nature of promises when embedded within an explicit monad as in Haskell, some equational laws specifying the expected interplay between the primitives used for creating and using promises and the primitives of the embedding monad.

Doing so, we shall not only understand to which extent promises indeed form a monad but also what properties of promises enforces asynchronism and concurrency. As a by-product, we shall see how the resulting formalized notion of promises, combined with monads, suffice for defining a rather well-featured interface for concurrent data flow programing which efficiency have already been demonstrated in the context of asynchronous control of synchronous sound processing.

In other words, we do not propose yet another asynchronous concurrent library. Instead, generalizing Haskell *async* library, we define, quite in the abstract, some generic notion of an *asynchronous concurrent extension* of a monad, its expected semantics being defined by means of equational laws relating the behavior of the new primitives with the underlying monad primitives.

The properties of these extensions, some concrete instance examples, and various uniformly derived function examples, illustrate throughout the meaning and relevance of such a equational axiomatization of asynchronous concurrent programing interface.

Overview of paper content. Our presentation is organized as follows.

We first review some basic notions around functors and monads in Section 2. Generalizing and abstracting the *async* library approach, we define in Section 3 a generic type class $MonadRef\ m$ composed of an abstract type $Ref_m\ a$ which elements are called monad references, built over the given monad m, together with two asynchronous primitives $fork :: m\ a \to m\ (Ref_m\ a)$ for creating monad references bound to forked monad actions, and $read :: Ref_m\ a \to m\ a$ for accessing, through these monad references, the values returned by the referenced forked actions.

The expected semantics of monad references is formalized by means of two series of equational laws describing the expected interplay between the monad primitives $return :: a \to m\ a$ and $bind :: m\ a \to (a \to m\ b) \to m\ b$ and the asynchronous primitives *fork* and *read*.

Described in Section 3, the first series of laws aim at capturing the basic semantics of monad references: how monad references are indeed bound to forked actions. Thanks to these laws, the monadic nature of promises can be investigated in the depth. Among other properties, the type function $m \circ Ref_m$ is proved to be a functor and, under some adequate restrictions, it is also a monad as shown in Section 4.

Described in Section 5, the second series of laws is more concerned with the asynchronous and concurrent nature of monad references. More precisely, some idempotency or commutation rules are stated for ensuring that *fork* actions are non blocking and *read* actions have no side effects but waiting for the associated forked actions to terminate.

As an illustration of is second series of laws, a number of instances of the *MonadRef* class, visibly not asynchronous nor concurrent but satisfying the first series of laws, are shown to be eventually ruled out by that second series.

Last, in Section 6, we show how the notion of monad references can be lifted to more complex data types such as monadic streams: a fact especially useful for sharing monadic stream contents without duplicating their underlying side effects. This eventually proves in the abstract that even though asynchronous concurrent primitives are fairly limited compared to general concurrent primitives, they nevertheless suffice, when combined with monad primitives and inductive type definitions, for defining a fairly generic interface for reactive and concurrent, synchronous or asynchronous, stream programing.

The general coherence and relevance of our proposal is is illustrated throughout by defining a simple valid extension of Haskell IO monad, a self-contained simplified version of the existing Haskell *async* library. Up to some technicalities, its is argued that even the existing Haskell *async* library itself also yields a valid instance of the monad reference class type.

A Haskell based presentation. Most concepts are presented, throughout this paper, by means of Haskell type classes which instances are requested to satisfy some number of equational laws. Compared to a purely theoretical approach, such a presentation comes with some overhead, but also an immediate benefit: it is directly applicable as demonstrated by two associated libraries developed both in Haskell and OCaml.

Throughout the paper, we shall also consider that two elements $a_1\ a_2 :: a$ of a given type a are *equal* when there are *indistinguishable* in any context of use in the sense that for any function $f :: a \to IO\ ()$ there is no observable difference between running $f\ \ a_1$ and $f\ \ a_2$ in some idealized IO monad. This means that either equality in a type a is (inductively and explicitly) defined in an instance $EQ\ a$ of the equality type class, or it is (co-inductively and implicitly) defined by contextual indistinguishability.

2 Preliminaries on monadic functors and monad actions

We review below the definition and some properties of functors and monads, following the programmer's point of view offered by the pioneer works of Moggi [9] and Wadler [11].

2.1. Functors. A (type) functor is a type function $m :: * \to *$ equipped with an *fmap* function as specified by the following class type:

```
class Functor m where
  fmap :: (a → b) → m a → m b
```

such that the following laws are satisfied:

$$m \equiv fmap\ id\ m \tag{1}$$

$$fmap\ g\ (fmap\ f\ \ m) \equiv fmap\ (g \circ f)\ m \tag{2}$$

for every $m :: m\ a$ every $f : a \to b$ and $g : b \to c$.

In other words, the function *fmap* extends to typed functions the function m over type. The first equation states that the image of the identity function by a functor shall be the identity. The second equation states that the image of the composition shall be the composition of the image.

2.2. Monads. A monad is a (type) functor $m :: * \rightarrow *$ equipped with two additional primitives *return* and *bind* as specified by the class type:

$$
\begin{array}{l}
\textbf{class } \mathit{Functor}\ m \Rightarrow \mathit{Monad}\ m \textbf{ where} \\
\quad \mathit{return} :: a \rightarrow m\ a \\
\quad (\ggg\!=) :: m\ a \rightarrow (a \rightarrow m\ b) \rightarrow m\ b
\end{array}
$$

such that the following equation are satisfied:

$$\mathit{return}\ a \ggg\!= f \equiv f\ a \tag{3}$$

$$m \ggg\!= \mathit{return} \equiv m \tag{4}$$

$$(m \ggg\!= f) \ggg\!= g \equiv m \ggg\!= (\lambda x \rightarrow f\ x \ggg\!= g) \tag{5}$$

for every $m :: m\ a$, $f :: a \rightarrow m\ b$ and $g :: b \rightarrow m\ c$, the infix operator $(\ggg\!=)$ named *bind* when used as a function. Elements of type $(m\ a)$ called *monad actions*.

The first and second equations state that, in some sense, *return* act as a *neutral element* for the bind, both on the left (3) and on the right (4). The third equation states that the bind operator is *associative* (5) in some sense. Later in the text, we may also denote by $m_1 \gg m_2$ the composition $m_1 \ggg\!= \lambda\_ \rightarrow m_2$.

2.3. Coherence property. Under such a presentation of monads, every monad instance shall also satisfy the following *coherence property*:

$$\mathit{fmap}\ f\ m \equiv m \ggg\!= (\mathit{return} \circ f) \tag{6}$$

for every $f :: a \rightarrow b$ and $m :: m\ a$, that states that the mapping function induced by the monad primitives equals the mapping function defined in the parent *Functor* class instance.

When m is a monad, the function $\lambda f\ m \rightarrow m \ggg\!= (\mathit{return} \circ f)$ indeed satisfies both functor laws. Equation (1) immediately follows from (4). Equation (2), which can be rephrased in terms of monad primitives by:

$$m \ggg\!= \mathit{return}\ (g \circ f) \equiv m \ggg\!= (\mathit{return} \circ f) \ggg\!= (\mathit{return} \circ g) \tag{7}$$

follows from (3) and (5).

2.4. Alternative syntax for binds. Haskell *do-notation* allows writing simpler composition of monad actions. Indeed, we may write:

$$\textbf{do}\ \{x_1 \leftarrow m_1; x_2 \leftarrow m_2; ...; x_{n-1} \leftarrow m_{n-1}; m_n\}$$

the variables x_1, x_2, ..., x_{n-1} possibly omitted when not used, in place for the bind series $m_1 \ggg\!= \lambda x_1 \rightarrow m_2 \ggg\!= \lambda x_2 \rightarrow ...m_{n-1} \ggg\!= \lambda x_{n-1} \rightarrow m_n$ with m_1, m_2, ..., m_n some monadic actions possibly depending on variables with strictly lower indices.

Such a notation has a clear flavor of imperative programing. Moreover, since action m_i possibly depends on the values returned by all actions m_j with $j < i$, it even seems that such a composition of action is necessarily evaluated from left to right. However, with Haskell lazy evaluation, this is not true in general unless the considered monad is *strict* as the IO Monad reviewed below.

2.5. The IO monad. For the reader not much familiar with monad programing, we review here some basic features of Haskell IO monad; a monad that allows pure functions to be used in communication with the real world.

The archetypal functions in the IO monad are $getChar :: IO\ \ Char$ and $putChar :: Char \rightarrow IO\ \ ()$ that respectively allows for getting the next character typed on the keyboard (*getChar*), or printing on the screen the character passed as argument (*putChar*).

As an exemple of a bind, there is the action $getChar \gg= putChar$ that gets the next typed character and prints it out. An important feature of monadic IO actions, as monad actions, is that they are *not* executed unless passed to the top level. This allows the definition of the following function:

$$\begin{aligned}&echo :: IO\ ()\\&echo = getChar \gg= putChar \gg= echo\end{aligned}$$

presumably infinite, which, only when executed, will repeatedly wait for a character to be typed on the standard input and will print it out on the standard output.

This illustrates the fact that, especially in a concurrent setting, monads, with *return* and *bind* functions that create and combine monad actions, are particularly well suited to define programs that will eventually be run (or forked) later on.

Another important aspect of the IO monad in Haskell is that it is a *strict* monad in the sense that, when executing a bind, the left monadic action is executed long enough for its argument to be given to the right function argument *before* the right argument starts to be evaluated. This contrasts significantly with Haskell principle of lazy evaluation but clearly allows a better control, or even any control at all, on IO scheduling.

3 Elementary monad references

We describe here the first half of our formalization of promises, a notion deriving from the fairly general notion of monad references. As we shall see in Section 4, this definition suffices for analyzing the monadic nature of promises.

However, nothing enforces these promises to behave neither asynchronously nor concurrently. Concurrent monad references, from which derive the usual notion of concurrent promises, shall be defined and studied later in Section 5.

3.1. Monad reference. Intendedly, a monad reference shall be a reference to a "running" monad action allowing to freely read the value returned by that action without repeating the side effect of the action. In terms of Haskell type class, we put:

$$\begin{aligned}&\textbf{class}\ Monad\ m \Rightarrow MonadRef\ \ m\ \textbf{where}\\&\quad\textbf{type}\ \ Ref_m :: * \rightarrow *\\&\quad fork :: m\ \ a \rightarrow m\ (Ref_m\ \ a)\\&\quad read :: Ref_m\ \ a \rightarrow m\ \ a\end{aligned}$$

where $\mathit{Ref}_m\ a$ is the type of references to running actions of type $m\ a$, $\mathit{fork}\ m$ is the action that shall fork the monadic action m and (immediately) return a reference to that action, and $\mathit{read}\ r$ is the action that shall (possibly wait for and) return the value returned by the action referenced by r.

3.2. Basic semantics laws. Every instance of the *MonadRef* class shall first satisfy the following laws:

$$(\mathit{fork}\ m) \gg\!= \mathit{read} \equiv m \tag{8}$$

$$\mathit{fork} \circ \mathit{read} \equiv \mathit{return} \tag{9}$$

$$(\mathit{fork}\ m) \gg\!= \lambda r \rightarrow \mathit{fork}\ (\mathit{read}\ r \gg\!= f) \equiv \mathit{fork}\ (m \gg\!= f) \tag{10}$$

for every $m :: m\ a$, $f :: a \rightarrow m\ b$.

These laws describe the expected interplay between forks and read with the underlying monad primitives. Law (8) states the basic semantics of forks and reads: reading a just forked action essentially behave like that action, side-effect included ! Law (9) states that forking a read essentially amount to return an equivalent reference. Law (10) states that forking a bind can be decomposed into two successive forks, provided the reference yield by the first fork is passed through as argument of the second one.

3.3. Default IO monad references. The simplest and truly concurrent instance of IO references one can define, thanks to mutable variables *MVar* and the native thread provided by *forkIO* in concurrent Haskell [6], is described by the following instance:

```
newtype MRef a = MRef (MVar a)
instance MonadRef (IO) where
  type Ref_IO = MRef
  fork m = do {v ← newEmptyMVar; forkIO (m >>= putMVar v);
               return (MRef v)}
  read (MRef v) = readMVar v
```

With this definition, one can review all expected rules and check to which extent they are satisfied.

Law (8) is satisfied thanks to the fact that the side effects happening when executing m are the same as the side effects happening when executing $\mathit{forkIO}\ m$. Law (10) validity essentially follows from the same reason, *forkIO* being non blocking and *readMVar* returning the expected value essentially as soon as it is available.

Law (9) is less obviously satisfied for there are indeed two distinct mutable variables created that are required to be equivalent when used under *MRef*. However, they both contains the same value, essentially at the same time. Since they can only be read in a non destructive way (via *readMVar*), these two encapsulated mutable variables can thus be replaced one with the other without observable difference.

It shall be clear that with Haskell *async* library, though more complex, we can define a similar valid *MonadRef* instance, with *Async* in place off *MVar*, *async* in place of *forkIO* and *wait* in place of *readMVar*. Again, for law (9) to be satisfied, the type *Async a* must be encapsulated in order to hide/disable its defined equality predicate.

3.4. A source of counter examples. We have stated some equational properties that shall be satisfied by instances of monad references, and we shall even state some more. Still, one may wonder how to check that an equality *is not* satisfied. The above instance of IO references shall be our main source of counter-examples. The reason for this is that the IO monad convey an implicit but rather strong notion of time based on IO events. More precisely, we have already mentioned that its bind is *strict* and some actions in the IO monad are blocking, such as *getChar*, while some others are not, such as *printChar c*. Two complex actions can thus be distinguished by the visible side-effects they may performed before being eventually blocked.

For instance, with IO references, one can observe that *fork m* is non blocking, regardless of the forked action m. This provides the following provable example of inequality that we shall use several times in the text. With $m_0 :: m\ (\mathit{Ref}_{IO}\ \ \mathit{Char})$ defined by $m_0 = \mathit{getChar} \gg\!= (\mathit{fork} \circ \mathit{return})$, we have

$$\mathit{fork}\ (m_0 \gg\!= \mathit{read}) \not\equiv m_0$$

eventhough both actions essentially return equivalent monad references.

Indeed, the action m_0 returns a reference towards the next typed character. But its blocks until that character is typed. The action $\mathit{fork}\ (m_0 \gg\!= \mathit{read})$ returns a similar reference since, by (8) and (4), it is equivalent to $\mathit{fork}\ (\mathit{getChar})$. However, it is non blocking since *fork* is non blocking

In other words, it is false that $\mathit{fork}\ (m \gg\!= \mathit{read}) \equiv m$ in general. However, in the next section, we shall see and use the fact that when $m = \mathit{fork}\ m'$ with $m' :: m\ a$ then such an equation does hold.

4 Properties of elementary monad references

In this section, we shall study the properties deriving from our equational definition of elementary monad references. We thus assume a type function $m :: * \to *$ with its functor instance *Functor m*, its monad instance *Monad m*, and its extension with monad references as a *MonadRef m* instance. This means that we assume there are the function *fmap*, *return*, *bind*, *fork* and *read* typed as described above and satisfying (1)–(6) for monad primitives, and laws (8)–(10) for monad references primitives.

4.1. Induced functor. Observe that, although $\mathit{Ref}_m :: * \to *$ is a type function, it cannot be a functor since there is no function that allows to create or read a monad reference without entering into the monad m. As well known by Haskell programmers, there is no general no way to go outside a monad.

However, we shall prove here that the composition $m \circ Ref_m$, that maps every type a to the type $m\ (Ref_m\ \ a)$ of monad actions returning monad references, is itself a functor. Indeed, there is the function *fmapRef* defined by

$$\begin{aligned}&fmapRef :: MonaRef\ \ m \Rightarrow (a \to b) \to m\ (Ref_m\ \ a) \to m\ (Ref_m\ \ b)\\ &fmapRef\ f\ \ m = m \gg\!= \lambda r \to fork\ (read\ r \gg\!= (return \circ f))\end{aligned}$$

and we have:

Theorem 1 *The type function $m \circ Ref_m$ with fmapRef as mapping function is a functor.*

Proof. (sketch of) The fact that *fmapRef* satisfies law (1) follows from (9) and standard monad laws. The fact that *fmapRef* satisfies law (2) follows from (10) and standard monad laws.

The reader may find surprising that law (8), though describing the basic semantics of forks and reads, is not mentioned here. It turns out that it has somehow already been used in order to simplify the definition of function *fmapRef* given here as shown in Lemma 3.

Remark. We could have put $fmapRef\ f\ \ m = fork\ (m \gg\!= read \gg\!= return \circ f)$ instead. But then, we would have $fmapRef\ id\ \ m = fork\ (m \gg\!= read)$ which, as shown at the end of Section 3, is distinct from m in the IO monad as soon as m is blocking. In other words, such an alternative definition fails to satisfy law (1).

4.2. Induced natural transformations. Functors m and $m \circ Ref_m$ are tightly related. Indeed, slightly abusing Haskell notations, we define the functor transformations

$$\begin{aligned}&Fork :: m \Rightarrow m \circ Ref_m\\ &Fork_a = fork :: m\ \ a \to Ref_m\ \ a\\ &Read :: m \circ Ref_m \Rightarrow m\\ &Read_a = \lambda m \to m \gg\!= read :: m\ (Ref_m\ \ a) \to m\ \ a\end{aligned}$$

and we have:

Theorem 2 *Both Fork and Read are natural transformation. More precisely, for every function $f :: a \to b$, within monad m, law (10) implies that:*

$$fmapRef\ f\ \ (fork\ \ m) \equiv fork\ (fmap\ f\ \ m) \tag{11}$$

for every action $m :: m\ \ a$, and law (8) implies that:

$$fmap\ f\ \ (m \gg\!= read) \equiv fmapRef\ f\ \ m \gg\!= read \tag{12}$$

for every action $m :: m\ (Ref_m\ \ a)$.

Moreover, functor m turns out to be a retract *of functor $m \circ Ref_m$, that is, $Read \circ Fork$ is the identity transformation, as immediately implied by (8).*

Remark. The reverse composition $\mathit{Fork} \circ \mathit{Read}$ is not the identity as observed in the IO monad taking again a blocking action m with $\mathit{fork}\ (m \gg\!= \mathit{read}) \not\equiv m$.

Altogether, the fact that m is a (strict) retract of $m \circ \mathit{Ref}_m$ says in particular that whatever one can do in the monad m without monad references, one can mimicked is in a compositional way under the fork essentially without any change of behavior. However, since m and $m \circ \mathit{Ref}_m$ are not isomorphic functors, there may be behaviors definable with monad references that cannot be defined without.

4.3. The possibility of a monad. We have shown that $m \circ \mathit{Ref}_m$ is a functor. Shall this functor be monadic ? Strictly speaking, this is not true as we shall see here by enumerating all possible definitions for returns and binds.

First, it shall be clear that, up to equivalent definitions, the unique possibility of a function *return* is defined by:

$$\begin{aligned}&\mathit{returnRef} :: \mathit{MonadRef}\ a \Rightarrow a \to m\ (\mathit{Ref}_m\ a)\\ &\mathit{returnRef} = \mathit{fork} \circ \mathit{return}\end{aligned}$$

for it is the unique uniformly defined function inhabiting its type. For the bind, as already suggested in the introduction, there are the following candidates:

$\mathit{bindRef} :: m\ (\mathit{Ref}_m\ a) \to (a \to m\ (\mathit{Ref}_m\ b)) \to m\ (\mathit{Ref}_m\ b)$

(a) $\mathit{bindRef}\ m\ f = \mathit{fork}\ (m \gg\!= \mathit{read} \gg\!= f \gg\!= \mathit{read})$

(b) $\mathit{bindRef}\ m\ f = m \gg\!= \lambda r \to \mathit{fork}\ (\mathit{read}\ r \gg\!= f \gg\!= \mathit{read})$

(c) $\mathit{bindRef}\ m\ f = m \gg\!= \mathit{read} \gg\!= \lambda a \to \mathit{fork}\ (f\ a \gg\!= \mathit{read})$

(d) $\mathit{bindRef}\ m\ f = m \gg\!= \mathit{read} \gg\!= f$

Of course, the fact that such a list shall be, up to equivalence, complete with respect to uniformly definable bind candidates, necessitates a proof. Let us just observe that we have at least enumerated all possible insertions of a fork into the possible series of binds. The IO monad instance somehow forces to respect functional dependencies in sequence, and adding additional forks and reads essentially yield equivalent candidates thanks to rules (8)–(10).

Lemma 3 *Bind candidates (a), (c) and (d) fail to satisfy the right unit monad law* (4) *in the IO monad instance.*

In any instance, the bind candidate (b) satisfies the right monad unit law (4)*, the monad associativity law* (5)*, as well as the coherence law* (6) *with respect to fmapRef, that is, with (b), we have:*

$$\mathit{fmapRef}\ f\ m \equiv \mathit{bindRef}\ m\ (\mathit{returnRef} \circ f) \tag{13}$$

for every $m :: m\ (\mathit{Ref}_m\ a)$ *and* $f :: a \to b$.

Moreover, while the bind candidate (b) fails to satisfy the left unit monad law (3) *in the IO monad instance, if we* restrict *to to functions of the form* $\mathit{fork} \circ f$ *some* $f :: a \to m\ b$*, then the bind candidate (b) also satisfies law* (3) *in arbitrary instances.*

In other words, the bind candidate (b) is a good candidate for us to prove that $m \circ \mathit{Ref}_m$ is our expected monad of promises *provided* we restrict ourselves to elements of $m\ (\mathit{Ref}_m\ a)$ of the form $\mathit{fork} \circ m$ for some monad action $m :: m\ a$.

4.4. The expected monad of promises. In Haskell, promises can thus be defined as an encapsulation of the subtype of $m\ (\mathit{Ref}_m\ a)$ of elements (equivalent with those) generated by forks. This can be encoded by defining the type:

$$\textbf{newtype}\ \mathit{Promise}\ m\ a = \mathit{Promise}\ \{\mathit{thePromise} :: m\ (\mathit{Ref}_m\ a)\}$$

controlling the way action are lifted into promises, and the way promises can eventuality be turned back into regular monad actions by:

```
$forkP :: MonadRef\ m \Rightarrow m\ a \rightarrow Promise\ m\ a$
$forkP = Promise \circ fork$
$readP :: MonadRef\ m \Rightarrow Promise\ m\ a \rightarrow m\ a$
$readP\ p = (thePromise\ p) \gg\!= read$
```

Then, by Theorem 1 we have the valid functor instance:

```
$\textbf{instance}\ MonadRef\ m \Rightarrow Functor\ (Promise\ m)\ \textbf{where}$
  $fmap\ f\ (Promise\ m) = Promise\ (fmapRef\ f\ m)$
```

as well as, as we shall soon see, the valid the monad instance:

```
$\textbf{instance}\ MonadRef\ m \Rightarrow Monad\ (Promise\ m)\ \textbf{where}$
  $return = Promise \circ returnRef$
  $(\gg\!=)\ (Promise\ m)\ f$
     $= Promise\ (bindRef\ m\ (thePromise \circ f))$
```

with bind candidate (b).

Indeed, a first lemma, easily proved by induction on the syntactic complexity of promises, gives:

Lemma 4 *Every promise built with the fonctions above is equivalent with a promise of the form* $\mathit{Promise}\ (\mathit{fork}\ m)$ *for some* $m :: m\ a$.

from which we deduce:

Theorem 5 *The above Monad instance is valid.*

Proof. By Lemma 3, we just have to prove that the monad left unit law (3) is satisfied which follows from Lemma 4.

Doing so, we eventually recover the monad promises defined in OCaml [8]. We even have:

Theorem 6 *Functor m and Promise m are isomorphic.*

Proof. Follows from Theorem 2 and Lemma 4 since, restricted to monad action of the form $\mathit{fork}\ m$ with $m :: m\ a$ we indeed have $\mathit{fork}\ (\mathit{fork}\ m \gg\!= \mathit{read}) \equiv \mathit{fork}\ m$ by law (8) therefore $\mathit{Promise} \circ \mathit{Fork}$ is the inverse of $\mathit{Read} \circ \mathit{thePromise}$.

5 Concurrent monad references

We aim at capturing asynchronous concurrent behaviors by means of the notion of monad references. However, as already mentioned, laws (8)–(10) fails to achieve such a goal. In this section, we shall add three additional laws that eventually complete our proposed axiomatization.

5.1. Pathological instances. Each of the following instances violates (at least) one specific property that asynchronous concurrent primitives shall satisfy.

Read effect-freeness (A). As a first example, the following instance, violates the intention that a monad reference should be freely readable, essentially with no side effects but waiting for the termination of the forked action.

```
instance MonadRef IO where
  type Ref_IO = IO
  fork = return
  read = id
```

Such an instance, that could be generalized to an arbitrary monad, is valid. Indeed, law (8) follows from (3), law (9) is immediate, and law (10) follows from (5). However, reading such a kind of reference just amounts to performing the referenced action therefore it has *arbitrary* side effects.

Non-blocking fork (B). As another example, despite the fact we said *fork* should be instantaneous, or at least non blocking, the following valid instance provide a counter example to that claim. Indeed, with $newMVar :: a \to m\ (MVar\ a)$ that creates a new mutable variable filled with its argument, we can put:

```
instance MonadRef IO where
  type Ref_IO = MRef
  fork m = m >>= (MRef ∘ newMVar)
  read (MRef v) = readMVar v
```

that is, just changing the definition of *fork* compared to the instance of IO references given in the previous section. Clearly, with $m = getChar$ then $fork\ m$ is blocking in the above instance.

Read independence (C). With a bit more of coding, the following instance is a more subtle counter-example to the intention that forking an action amounts to execute it. Indeed, with:

```
instance MonadRef IO where
  type Ref_IO a = MRef (Either (IO a) a)
  fork m = MRef (newMVar (Left m))
  read (MRef v) = do { c ← takeMVar v;
    a ← case c of { Left m → m; Right a → return a };
    putMVar v (Right a); return a }
```

a forked action is executed only when its associated reference is red for the first time.

5.2. Concurrency laws. The following second series of laws, that shall also be satisfied by instance of *MonadRef*, rules out the above pathological instances as invalid:

$$\mathit{read}\ r \equiv \mathit{read}\ r \gg \mathit{read}\ r \tag{14}$$

$$\begin{aligned}&\mathit{fork}\ m_1 \ggg\!= \lambda r_1 \rightarrow (\mathit{fork}\ m_2 \ggg\!= \lambda r_2 \rightarrow \mathit{return}\ (r_1, r_2))\\ &\qquad\equiv \mathit{fork}\ m_2 \ggg\!= \lambda r_2 \rightarrow (\mathit{fork}\ m_1 \ggg\!= \lambda r_1 \rightarrow \mathit{return}\ (r_1, r_2))\end{aligned} \tag{15}$$

$$\begin{aligned}&\mathit{read}\ r_1 \ggg\!= \lambda x_1 \rightarrow (\mathit{read}\ r_2 \ggg\!= \lambda x_2 \rightarrow \mathit{return}\ (x_1, x_2))\\ &\qquad\equiv \mathit{read}\ r_2 \ggg\!= \lambda x_2 \rightarrow (\mathit{read}\ r_1 \ggg\!= \lambda x_1 \rightarrow \mathit{return}\ (x_1, x_2))\end{aligned} \tag{16}$$

for every monad reference $r\ r_1\ r_2 :: \mathit{Ref}_m\ a$ and monad action $m_1\ m_2 :: m\ a$.

Law (14) states that reference reads are idempotent therefore with side-effects occurring at most with the execution of their first occurrence. Our pathological instance (A) is ruled out by such a rule for there are, in this instance, plenty of non idempotent read actions. However, both pathological instances (B) and (C) satisfy such a law.

By stating that fork actions commute, law (15) stresses on the instantaneity of fork actions. This law rules out the pathological instance (B). Indeed, a blocking action such as *getChar* visibly does not commute with a non blocking but observable action such as *printChar c*. However, the pathological instance (C) still satisfy such a law.

Last, by stating that read actions commute, law (16) aims at capturing the independency of the execution of a forked actions with respect to the associated reads. Read action shall truly have no other side-effect but waiting for the forked action to be completed. Our last pathological instance (C) is eventually ruled out by such a rule as shown by forking both a blocking IO action and an observable non blocking one, and observing the non commutation of their resulting reads.

In other words, these additional rules have ruled out all pathological instances we could think of. This increase our confidence in the fact that they eventually form a complete axiomatization of asynchronous concurrent behaviors. Of course, there is no guarantee there are no other pathological instances. Even more, in which sense both series of laws (8)–(10) and (14)–(16) should be complete for defining asynchronous concurrency is not clear at all.

5.3. Validity in the IO monad. In the IO instance of monad references defined in Section 3, law (14) follows from the fact that the action *readMVar* is non destructive.

Law (15) is perhaps the most debatable for it may wrongly suggest that the side effect of action m_1 and m_2 commute. This is not true. In the concurrent framework of Haskell, these side effect are executed in parallel therefore, up to the possible non determinism induced by that parallelism, forking m_1 right before m_2 or m_2 right before m_2 essentially produces the same side-effect.

Law (16) shall make no difficulty to be accepted as valid since reads essentially wait for termination of (parallel) forked actions. Waiting for the termination of one action and then another just amount to waiting for the termination of both.

Remark. Of course, concurrency yields non determinism as made explicit by the commutations of forks. An example of non determinism on outputs is given by any of the following equivalent programs:

$$\begin{aligned}&\mathit{fork}\ (\mathit{putChar}\ \texttt{'a'}) \gg \mathit{fork}\ (\mathit{putChar}\ \texttt{'b'})\\ &\mathit{fork}\ (\mathit{putChar}\ \texttt{'b'}) \gg \mathit{fork}\ (\mathit{putChar}\ \texttt{'a'})\end{aligned}$$

that print non deterministically either `ab` or `ba`. An exemple of non determinism on inputs is given by any of the following equivalent programs:

$$\begin{aligned}&\mathit{fork}\ (\mathit{getChar}) \ggg\!= \lambda r_1 \to \mathit{fork}\ (\mathit{getChar}) \ggg\!= \lambda r_2 \to \mathit{read}\ r_1 \ggg\!= \mathit{printChar}\\ &\mathit{fork}\ (\mathit{getChar}) \ggg\!= \lambda r_2 \to \mathit{fork}\ (\mathit{getChar}) \ggg\!= \lambda r_1 \to \mathit{read}\ r_1 \ggg\!= \mathit{printChar}\end{aligned}$$

that, `ab` on the input, print non deterministically either `a` or `b`.

5.4. More concurrent primitives. In a concurrent setting, a running monad action may not be terminated, so we may just *try* to read its returned value. Also, between two running monad actions, one may terminate before the other hence we may wait for the *earliest* terminated one.

These two aspects, that cannot derive from the primitives defined so far, can be depicted by the following type class:

```
class MonadRef m ⇒ MonadRefPlus m where
  tryRead :: Ref_m a → m (Maybe a)
  parRead :: Ref_m a → Ref_m b → m (Either a b)
```

where *tryReadRef* r is the action that shall immediately return nothing if the referenced action is not terminated or just its returned value otherwise and *parReadRef* r_1 r_2 is the action that shall return the value of the soonest terminated referenced action or, in the case both actions are already terminated or are terminating at the same time, any of the returned value.

In the IO monad, such a (more) concurrent extension of monad references can be defined by:

```
instance MonadRefPlus IO where
  tryRead (MRef v) = tryReadMVar v
  parRead r1 r2 = do
    { v ← newEmptyMVar;
      forkIO (read r1 >>= (tryPutMVar v) ∘ Left >> return ());
      forkIO (read r2 >>= (tryPutMVar v) ∘ Right >> return ());
      readMVar v }
```

It might be worth aiming at axiomatizing the behavior of these newly introduced primitives. For instance, one may expect to have:

- ▷ $\mathit{fork}\ m \ggg\!= \mathit{tryRead} \equiv m \ggg\!= \mathit{return}$ when m is instantaneous,
- ▷ $\mathit{fork}\ m \ggg\!= \mathit{tryRead} \equiv \mathit{return}\ \mathit{Nothing}$ when m is not instantaneous.

However, these laws seem to be difficult to be enforced at runtime and, at compile time, they require some typing of action duration typing that is not available.

More generally, the possibility of defining an equational theory for these new primitives bumped into the lack of a denotional semantics for modeling time. For instance, we cannot describe the property that between to monadic actions, one is finishing before the other unless they both block endlessly.

6 Concurrent structures and references

So far, we have only defined references to running monad actions. We aim now at extending monad references to generalized monad actions, that is, structures of nested monad actions. Eventhough this can easily be generalized to more complex structure, we shall simply review here the following case of monadic streams, a structure that nevertheless suffices for eventually extending asynchronous concurrent programming to concurrent data-flow programming.

6.1. Monad streams. Monad streams are defined by the following inductive data type:

```
data Stream m a = Stream {next :: m (Maybe (a, Stream m a))}
```

In other words, a monad stream is essentially defined as a monad action that either return nothing when the stream terminates, or just a value and the action defining the continuation of that stream.

Discussing in the depth about such a type constructor goes out of the scope of the present paper. Let us just mentioned that it is somehow fairly popular among Haskell programmers. One of its generalization constitutes the core of the *Conduit* library developed by Michael Snoyman. It has also recently been used in this simple form for realtime audio processing and control [5].

6.2. Derived functor instance. As a typical example of monad stream programing, there is the following functor instance.

```
instance Monad m ⇒ Functor (Stream m) where
  fmap f (Stream m) = Stream $ do {c ← m;
    case c of {Nothing → return Nothing;
      Just (a, sc) → return $ Just (f a, fmap f sc)}}
```

Every function $fmap\ f :: Monad\ m \Rightarrow Stream\ m\ a \to Stream\ m\ b$ is an archetypal example of a synchronous function over monadic streams.

6.3. Horizontal monoid structure. There is the following monoid instance that essentially lifts to monadic stream the (free) monoid encoded by the list data type.

```
instance Monad m ⇒ Monoid (Stream m a) where
  mempty = Stream (return Nothing)
  (◊) (Stream m) s = Stream $ do
    {c ← m; case c of {Nothing → next s;
      Just (a, sc) → return $ Just (a, sc ◊ s)}}
```

where the neutral element *mempty* is the (immediately) empty streams and $(\Diamond)$ is the concatenation function.

In a concurrent and reactive context, the horizontal concatenation is of little use unless its first argument is a constant and thus acts as a delay/buffering. We shall see below, in link with monad references, a much more interesting monoid and related monad instance defined for monad streams.

6.4. Monad stream references. Whenever a monad has references, there is a generalized notion of references applicable to streams built on that monad that easily derived from both notions.

Indeed, a reference to a monad stream can just be defined by replacing the monad type constructor in the definition of streams by the monad reference type constructor.

```
type StreamRef_m = Stream Ref_m
forkStream :: MonadRef m ⇒ Stream m a → m (StreamRef_m a)
forkStream = fork (evalAndFork s) ≫= return ∘ Stream
  where
    evalAndFork (Stream m) = m ≫= mapM
      (λ(a, sc) → do {rc ← fork (evalAndFork sc); return (a, Stream rc)})
readStream :: MonadRef m ⇒ StreamRef_m a → m (Stream m a)
readStream (Stream r) = return ∘ Stream $ read r ≫=
  mapM (λ(a, rc) → return (a, readStream rc))
```

A major application of *forkStream* and *readStream* is the possibility to share the content of a stream without duplicating its side effect. Such a possibility is especially useful in reactive on-the-fly data-flow programming [5].

Back to the study of reference properties, one can prove, that:

Lemma 7 *For every $s :: Stream\ m$ we have:*

$$forkStream\ s \ggg\!= readStream \equiv return\ s$$
$$forkStream \circ readStream \equiv return$$

In other words, with monadic stream references defined as above, the first two laws (8)– (9) lift to the case of monadic stream references.

For the third property (10), we eventually need to equip monadic stream with an adequate monad structure. This monad goes via the definition of the following vertical monoid structure on streams.

6.5. Vertical monoid structure. Thanks to *parRead* one can define the merge of two monadic streams by:

```
merge :: MonadRefPlus m ⇒ Stream m a → Stream m a → Stream m a
merge s₁ s₂ = Stream $ do
  {r₁ ← forkT s₁; r₂ ← forkT s₂; return (next $ mergeRef r₁ r₂)}
```

with

```
mergeRef :: MonadRefPlus m ⇒
  Stream Ref_m a → Stream Ref_m a → Stream m a
mergeRef (Stream r_1) (Stream r_2) = Stream $ do
  { c ← parRead r_1 r_2; case c of {
    Left Nothing → next $ readT (Stream r_2);
    Right Nothing → next $ readT (Stream r_1);
    Left (Just (a, src1)) → return $ Just (a, mergeRef src1 sr_2);
    Right (Just (a, src2)) → return $ Just (a, mergeRef sr_1 src2)}}
```

Then, up to the possible non determinism yields by *parRead*, the type *stream m a* of monadic streams equipped with *merge* is essentially a commutative monoid with the empty stream *mempty* as neutral element.

6.6. Derived stream monad. Thanks to such a vertical monoid structure, we have the following valid monad instance:

```
instance MonadRef m ⇒ Monad (Stream m) where
  return a = (Stream ∘ return ∘ Just) (a, mempty)
  (>>=) (Stream m) f = Stream $ do
    { c ← m; case c of
     { Nothing → return Nothing;
       Just (a, mc) → next $ merge (f a) (mc >>= f)}}
```

There, the flattening operation essentially amounts to inductively merge (sub)monadic streams from the moment they appear.

Lemma 8 *For every stream s :: Stream m a and function $f :: a \to$ Stream m b, we have:*

$$\mathit{forkStream}\ (s \gg\!= f) \equiv \mathit{forkStream}\ s \gg\!= \lambda r \to \mathit{forkStream}\ (\mathit{readStream}\ r \gg\!= f)$$

In other words, the third monad reference law (10) lifts to monad stream references.

This monad instance of *Stream m* is also an extension of the monad *m* in the sense that, with:

```
liftStream :: m a → Stream m a
liftStream m = Stream $ do { a ← m; return $ Just (a, emptyStream)}
```

we have:

Lemma 9 *Function liftStream is a natural embedding of m into Stream m with:*

$$\begin{aligned} \mathit{liftStream} \circ \mathit{return} &\equiv \mathit{return} \\ \mathit{liftStream}\ (m \gg\!= f) &\equiv \mathit{liftStream}\ m \gg\!= \mathit{liftStream} \circ f \end{aligned}$$

for every action $m :: m\ a$ and function $f :: a \to m\ b$.

6.7. Generalization to monadic structure. The above treatment of monadic streams seems to fit the fairly general notion of monadic structure oine can define with the class type by:

$$\begin{array}{l}\textbf{class}\ (\mathit{ConcurrentRef}\ m, \mathit{Monad}\ (t\ m)) \Rightarrow \mathit{MonadDataRef}\ t\ m\ \textbf{where}\\ \quad \mathit{forkT} :: t\ m\ a \to m\ (t\ \mathit{Ref}_m\ a)\\ \quad \mathit{readT} :: t\ \mathit{Ref}_m\ a \to m\ (t\ m\ a)\end{array}$$

where, in any instance, primitives *forkT* and *readT* are required to satisfy the following laws:

$$(\mathit{forkT}\ s) \gg\!= \mathit{readT} \equiv \mathit{return}\ s \tag{17}$$

$$\mathit{forkT} \circ \mathit{readT} \equiv \mathit{return} \tag{18}$$

$$\mathit{forkT}\ (s \gg\!= f) \equiv (\mathit{forkT}\ s) \gg\!= \lambda r \to \mathit{forkT}\ (\mathit{readT}\ r \gg\!= f) \tag{19}$$

for every $s :: \mathit{RefT}_m\ a$, $f :: a \to \mathit{RefT}_m\ b$ for the basic semantics. Indeed, thanks to Lemmas 7 and 8 the is the following valid instance:

$$\begin{array}{l}\textbf{instance}\ \mathit{ConcurrentRef}\ m \Rightarrow \mathit{Stream}\ m\ a\ \textbf{where}\\ \quad \mathit{forkT} = \mathit{forkStream}\\ \quad \mathit{readT} = \mathit{readStream}\end{array}$$

It is probably the case that such a construction can be generalized to arbitrary monadic version of inductive types, defined as least fixpoints of positive type functions of the form $m \circ F : * \to *$ built out positive type functions $F : * \to *$ used to define inductive types. However, such a study goes out of the scope of the present paper.

6.8. More parallelism. So far, we can fork one monad action, or a stream of nested monad actions. One may wonder if such a fork can be generalized to other structures such as lists, or, more generally, traversable structures. Actually, this can easily be done by:

$$\begin{array}{l}\mathit{forkAll} :: (\mathit{Traversable}\ t, \mathit{MonadRef}\ m) \Rightarrow t\ (m\ a) \to m\ (t\ (\mathit{Ref}_m\ a))\\ \mathit{forkAll} = \mathit{mapM}\ \mathit{fork}\end{array}$$

The question then becomes, how to handle the resulting structure of monad references. One possibility is to uniformly define:

$$\begin{array}{l}\mathit{sortRefs} :: (\mathit{Traversable}\ t, \mathit{ConcurrentRef}\ m) \Rightarrow t\ (\mathit{Ref}_m\ a) \to\\ \mathit{Stream}\ m\ a\ \mathit{sortRefs} = \mathit{foldMap}\ (\mathit{liftStream} \circ \mathit{read})\end{array}$$

that turns a traversable structure of monad references into the monad stream of values returned by the referenced actions *ordered* by termination time. In other words, *sortRefs* generalizes *parRead* to arbitrary traversable structures. Moreover, functions *forkAll* and *sortRefs* share with monad reference primitives *fork* and *read* their safety with no possible deadlock.

Of course, such a generic instance suffers from a rather severe drawback: its complexity in terms of call to *parRead* therefore in number of *fork* is likely to be quadratic in the size of the traversable structure.

With concurrent Haskell, this is not a necessity as shown by the following direct implementation of *sortRefs* in the IO monad:

```
    $sortRefs_{IO} :: Traversable\ t \Rightarrow t\ (Ref_{IO}\ \ a) \rightarrow IO\ (Stream\ IO\ a)$
    $sortRefs_{IO}\ \ t = \mathbf{do}\ \{ v \leftarrow newEmptyMVar;$
      $mapM\_\ (\lambda r \rightarrow forkIO\ (read\ r \gg= putMVar\ v))\ t;$
      $return\ \$\ mvarToStream\ v\ (length\ t) \}$
  $\mathbf{where}$
    $mvarToStream\ \_\ 0 = mempty$
    $mvarToStream\ v\ n = Stream\ \$\ \mathbf{do}$
      $\{ a \leftarrow takeMVar\ v; return\ \$\ Just\ (a, mvarToStream\ v\ (n-1)) \}$
```

with a linear number of forks.

In other words, despite the many and somewhat unexpected programming possibilities offered by asynchronous concurrency, illustrated among other things by monad stream references, asynchronous concurrency does not offer as much programming possibilities as a more general concurrent programming framework. The robustness and safety offered by asynchronous concurrency comes with a price.

7 Related works and conclusion

The study proposed here started as an attempt to clarify the underlying properties of an existing and somewhat adhoc but rather succesfull experiment of audio processing and control programming in Haskell [5]. As such, it is a bit of a stand alone approach that is a priori not much related with former theoretical investigations. Our proposal can nevertheless be seen as a equational formalization of the semantics of existing *async* libraries. To the best of our knowledge, neither in OCaml nor in Haskell such an axiomatization has yet been attempted.

Such a formalization relies on a fairly generic notion of a monad extension, the first series of laws describing how to go back and forth between the underlying monad m and its extension $m \circ Ref_m$ via the retraction pair of natural transformation it induces (Theorem 2). It is only the second series of laws that deals with concurrent semantics aspects.

Quite strikingly, such a kind of relationship reappears again between monadic structure and monadic structure references and, to some extent, between traversable structures of monad actions or monad references. The underlying general category theoretic schema is probably worth being studied more in the depth.

Compared to existing techniques, that generally amounts to combining two existing monads [9, 2], our monad extension proposal does not seem to be of that kind. Indeed, the type function Ref_m enriching the available types is even not a functor. The type function $m \circ Ref_m$ is a functor, but a monad only when restricted to forked monad actions. Moreover, as opposed monad combination techniques, the resulting extended monad is not a new monad but the original monad merely extended by new primitives. Instead, one might think that our notion of monad extension is connected with the notion algebraic effects [1]. However, we offer yet no operational semantic modeling and some typical features of algebraic effects are not yet made explicit in our proposal.

As already mentioned in the text, the lack of a model of passing time prevents us from formally defining the semantics of the *parRead* function. The absence of a formal operational semantics also prevents us from even stating any kind of completeness results for our proposed axiomatization. However, a pure denotational approach might still be possible following the recent proposal of a timed extension of Scott domains [4].

References

1. An introduction to algebraic effects and handlers. invited tutorial paper. *Electronic Notes in Theoretical Computer Science*, 319:19 – 35, 2015. The 31st Conference on the Mathematical Foundations of Programming Semantics (MFPS XXXI).
2. F. Dahlqvist, L. Parlant, and A. Silva. Layer by layer - combining monads. In *Theoretical Aspects of Computing (ICTAC)*, pages 153–172, 2018.
3. Robert H. Halstead, Jr. Multilisp: A language for concurrent symbolic computation. *ACM Trans. Program. Lang. Syst.*, 7(4):501–538, 1985.
4. D. Janin. Spatio-temporal domains: an overview. In *Int. Col. on Theor. Aspects of Comp. (ICTAC)*, volume 11187 of *LNCS*, pages 231–251. Springer-Verlag, 2018.
5. D. Janin. Screaming in the IO monad. In *ACM Work. on Functional Art, Music, Modeling and Design (FARM)*. ACM, 2019.
6. S. Peyton Jones, A. Gordon, and S. Finne. Concurrent haskell. In *Principles of Programing Languages (POPL)*, New York, 1996. ACM.
7. S. Marlow. *Parallel and Concurrent Programming in Haskell.* O'Reilly, 2013.
8. Y. Minsky, J. Hickey, and A. Madhavapeddy. *Real World Ocaml: Functional programming for the masses.* O'Reilly, 2013.
9. E. Moggi. A modular approach to denotational semantics. In *Category Theory and Computer Science (CTCS)*, volume 530 of *LNCS*. Springer-Verlag, 1991.
10. J. H. Reppy. *Concurrent Programming in ML.* Cambridge University Press, 1999.
11. P. Wadler. Comprehending monads. In *Conference on LISP and Functional Programming*, LFP '90, pages 61–78, New York, 1990. ACM.

A Omitted proofs

1.1. Proof of Theorem 1.

Proof. Let $m :: m\ (\mathit{Ref}\ \ a)$ be an arbitrary monad action returning a monad reference and let $f :: a \to b$ and $g :: b \to c$. For (1), we have:

$$\begin{aligned}
&\mathit{fmap}\ \mathit{id}\ m \\
&= m \ggg\!= (\lambda r \to \mathit{fork}\ (\mathit{read}\ r) \ggg\!= \mathit{return}) \\
&\equiv m \ggg\!= (\mathit{fork} \circ \mathit{read}) && \mathit{by}\ (4) \\
&\equiv m \ggg\!= \mathit{return} && \mathit{by}\ (9) \\
&\equiv m && \mathit{by}\ (4)
\end{aligned}$$

For (2), we have:

$$\begin{aligned}
&\mathit{fmapRef}\ (g \circ f)\ m \\
&= m \ggg\!= \lambda r \to \mathit{fork}\ (\mathit{read}\ r \ggg\!= (\mathit{return} \circ g \circ f)) \\
&\equiv m \ggg\!= \lambda r \to \mathit{fork}\ (\mathit{read}\ r \ggg\!= (\mathit{return} \circ f) \\
&\qquad\qquad\qquad\qquad \ggg\!= (\mathit{return} \circ g)) && \mathit{by}\ (6),\ (2) \\
&\equiv m \ggg\!= \lambda r \to (\mathit{fork}\ (\mathit{read}\ r \ggg\!= (\mathit{return} \circ f)) \\
&\qquad \ggg\!= \lambda r_1 \to \mathit{fork}\ (\mathit{read}\ r_1 \ggg\!= (\mathit{return} \circ g))) && \mathit{by}\ (10) \\
&\equiv m \ggg\!= \lambda r \to \mathit{fork}\ (\mathit{read}\ r \ggg\!= (\mathit{return} \circ f)) \\
&\quad \ggg\!= \lambda r_1 \to \mathit{fork}\ (\mathit{read}\ r_1 \ggg\!= (\mathit{return} \circ g)) && \mathit{by}\ (5) \\
&= \mathit{fmapRef}\ m\ f \\
&\quad \ggg\!= \lambda r_1 \to \mathit{fork}\ (\mathit{read}\ r_1 \ggg\!= (\mathit{return} \circ g)) \\
&= \mathit{fmapRef}\ g\ (\mathit{fmap}\ f\ m)
\end{aligned}$$

□

1.2. Proof of Theorem 2.

Proof. Let $f : a \to b$ and let $m :: m\ a$. We first prove (11). We have:

$$\begin{aligned}
&\mathit{fmapRef}\ f\ (\mathit{fork}\ m) \\
&= \mathit{fork}\ m \ggg\!= \lambda r \to \mathit{fork}\ (\mathit{read}\ r \ggg\!= (\mathit{return} \circ f)) \\
&\equiv \mathit{fork}\ (m \ggg\!= \mathit{return} \circ f) && \mathit{by}\ (10) \\
&\equiv \mathit{fork}\ (\mathit{fmap}\ f\ m) && \mathit{by}\ (6)
\end{aligned}$$

Let instead $m :: m\ (\mathit{Ref}\ \ m\ a)$. We prove (12). We have:

$$\begin{aligned}
&\mathit{fmapRef}\ f\ m \ggg\!= \mathit{read} \\
&= m \ggg\!= \lambda r \to \mathit{fork}\ (\mathit{read}\ r \ggg\!= (\mathit{return} \circ f)) \ggg\!= \mathit{read} \\
&\equiv m \ggg\!= \lambda r \to (\mathit{fork}\ (\mathit{read}\ r \ggg\!= (\mathit{return} \circ f)) \ggg\!= \mathit{read}) && \mathit{by}\ (5) \\
&\equiv m \ggg\!= \lambda r \to (\mathit{read}\ r \ggg\!= (\mathit{return} \circ f)) && \mathit{by}\ (8) \\
&\equiv m \ggg\!= \mathit{read} \ggg\!= (\mathit{return} \circ f) && \mathit{by}\ (5) \\
&= \mathit{fmap}\ f\ (m \ggg\!= \mathit{read}) && \mathit{by}\ (6)
\end{aligned}$$

Now, let us prove that $\mathit{Read} \circ \mathit{Fork} = \mathit{Id}$. Let again $m :: m\ a$. We have:

$$\begin{aligned}
&(\mathit{Read}_a \circ \mathit{Fork}_a)\ m) \\
&= \mathit{fork}\ m \gg\!= \mathit{read} \\
&\equiv m && \textit{by } (8)
\end{aligned}$$

□

1.3. Proof of Lemma 3.

Proof. Law (4) says that $\mathit{bindRef}\ m\ \mathit{returnRef}$ shall be equivalent to m.

With $m_0 = \mathit{getChar} \gg\!= \mathit{fork} \circ \mathit{return}$ in the IO monad, law (4) can't be satisfied by (a) since m_0 is blocking while $\mathit{bindRef}\ m_0\ \mathit{returnRef}$ is not.

Taking instead $m_1 = \mathit{fork}\ m_0$ law (4) can't be satisfied by (c) or (d) since now m_1 is non blocking while $m_1 \gg\!= \mathit{read}$ is blocking since, by (8), it is equivalent to m_0.

Let us consider from now on and throughout this proof candidate (b). We first prove that (3) fails in the IO monad. For such a purpose, let $f = \mathit{const}\ m_0$ with m_0 as defined above in the IO monad. Then we have

$$\mathit{bindRef}\ (\mathit{return}\ r)\ f = \mathit{return}\ r \gg\!= \lambda r' \to \mathit{fork}\ (\mathit{read}\ r' \gg\!= f \gg\!= \mathit{read})$$

which is non blocking while

$$f\ r = m_0$$

which is blocking. This proves the negative claim.

Let us prove that law (4) is satisfied by (b). Let $m :: m\ a$. We have

$$\begin{aligned}
&\mathit{bindRef}\ m\ \mathit{returnRef} \\
&= m \gg\!= \lambda r \to \\
&\qquad \mathit{fork}\ (\mathit{read}\ r \gg\!= \mathit{fork} \circ \mathit{return} \gg\!= \mathit{read}) \\
&\equiv m \gg\!= \lambda r \to \mathit{fork}\ (\mathit{read}\ r \gg\!= \mathit{return}) && \textit{by } (8) \\
&\equiv m \gg\!= \mathit{fork} \circ \mathit{read}) && \textit{by } (4) \\
&\equiv m \gg\!= \mathit{return} && \textit{by } (9) \\
&\equiv m && \textit{by } (4)
\end{aligned}$$

Let us prove now law (5) for (b). Let $m :: m\ (\mathit{Ref}\ \ m\ \ a)$, $f :: a \to m\ (\mathit{Ref}\ \ m\ \ b)$ and $g :: b \to m\ (\mathit{Ref}\ \ m\ \ c)$. We have

$$
\begin{array}{ll}
\mathit{bindRef}\ (\mathit{bindRef}\ \ m\ f)\ g & \\
\quad = (m \gg\!= \lambda r \to \mathit{fork}\ (\mathit{read}\ \ r \gg\!= f \gg\!= \mathit{read})) & \\
\qquad \gg\!= \lambda r' \to \mathit{fork}\ (\mathit{read}\ \ r' \gg\!= g \gg\!= \mathit{read}) & \\
\quad \equiv m \gg\!= \lambda r \to (\mathit{fork}\ (\mathit{read}\ \ r \gg\!= f \gg\!= \mathit{read}) & \\
\qquad \gg\!= \lambda r' \to \mathit{fork}\ (\mathit{read}\ \ r' \gg\!= g \gg\!= \mathit{read})) & \mathit{by}\ (5) \\
\quad \equiv m \gg\!= \lambda r \to \mathit{fork}\ (\mathit{read}\ \ r \gg\!= f \gg\!= \mathit{read} \gg\!= g \gg\!= \mathit{read}) & \mathit{by}\ (10) \\
\quad \equiv m \gg\!= \lambda r \to \mathit{fork}\ (\mathit{read}\ \ r \gg\!= f \gg\!= \lambda r \to (\mathit{read}\ \ r \gg\!= g \gg\!= \mathit{read})) & \mathit{by}\ (5) \\
\quad \equiv m \gg\!= \lambda r \to \mathit{fork}\ (\mathit{read}\ \ r \gg\!= f & \\
\qquad \gg\!= \lambda r \to (\mathit{fork}\ (\mathit{read}\ \ r \gg\!= g \gg\!= \mathit{read}) \gg\!= \mathit{read})) & \mathit{by}\ (8) \\
\quad \equiv m \gg\!= \lambda r \to \mathit{fork}\ (\mathit{read}\ \ r \gg\!= f & \\
\qquad \gg\!= \lambda r \to \mathit{fork}\ (\mathit{read}\ \ r \gg\!= g \gg\!= \mathit{read}) \gg\!= \mathit{read}) & \mathit{by}\ (5) \\
\quad \equiv m \gg\!= \lambda r \to \mathit{fork}\ (\mathit{read}\ \ r \gg\!= (\lambda a \to f\ \ a & \\
\qquad \gg\!= \lambda r \to \mathit{fork}\ (\mathit{read}\ \ r \gg\!= g \gg\!= \mathit{read})) \gg\!= \mathit{read}) & \mathit{by}\ (5) \\
\quad = m \gg\!= \lambda r \to \mathit{fork}\ (\mathit{read}\ \ r \gg\!= (\lambda a \to \mathit{bindRef}\ \ (f\ \ a)\ \ g) \gg\!= \mathit{read}) & \\
\quad = \mathit{bindRef}\ \ m\ (\lambda a \to \mathit{bindRef}\ \ (f\ \ a)\ \ g) &
\end{array}
$$

Last, we prove the coherence law (6) with (b). Let $m :: m\ (\mathit{Ref}_m\ \ a)$ and let $f : a \to b$. We have:

$$
\begin{array}{ll}
\mathit{bindRef}\ \ m\ (\mathit{returnRef} \circ f) & \\
\quad = m \gg\!= \lambda r \to \mathit{fork}\ (\mathit{read}\ \ r \gg\!= (\mathit{fork} \circ \mathit{return} \circ f) \gg\!= \mathit{read}) & \\
\quad \equiv m \gg\!= \lambda r \to \mathit{fork}\ (\mathit{read}\ \ r \gg\!= (\lambda a \to \mathit{fork}\ (\mathit{return}\ (f\ \ a)) \gg\!= \mathit{read})) & \mathit{by}\ (5) \\
\quad \equiv m \gg\!= \lambda r \to \mathit{fork}\ (\mathit{read}\ \ r \gg\!= (\mathit{return} \circ f)) & \mathit{by}\ (8) \\
\quad = \mathit{fmapRef}\ \ f\ \ m &
\end{array}
$$

Finally, still with candidate (b), assume $a :: m\ a$ and $f :: a \to m\ (\mathit{Ref}_m\ \ b)$ with f of the form $f = \mathit{fork} \circ g$ with $g :: a \to m\ b$. Then we have:

$$
\begin{array}{ll}
\mathit{bindRef}\ (\mathit{returnRef}\ \ a)\ f & \\
\quad = (\mathit{fork} \circ \mathit{return})\ a \gg\!= \lambda r' \to \mathit{fork}\ (\mathit{read}\ \ r' \gg\!= \mathit{fork} \circ g \gg\!= \mathit{read}) & \\
\quad \equiv \mathit{fork}\ (\mathit{return}\ \ a \gg\!= \mathit{fork} \circ g \gg\!= \mathit{read}) & \mathit{by}\ (10) \\
\quad \equiv \mathit{fork}\ (\mathit{fork}\ \ (g\ \ a) \gg\!= \mathit{read}) & \mathit{by}\ (3) \\
\quad \equiv \mathit{fork}\ \ (g\ \ a) & \mathit{by}\ (8) \\
\quad = f\ \ a &
\end{array}
$$

which prove that, indeed, law (3) is satisfied when restricting to forked monad actions in type $m\ (\mathit{Ref}_m\ \ b)$.

1.4. Proof of Lemma 4.

Proof. Clearly, function *forkP* as well as *Promise* ∘ *returnRef* creates promises of the desired form. It remains to show that, up to equivalence, such a form is preserved by the bind we have defined over promises. This is done by induction on the syntactic complexity of its arguments.

Let $p :: \mathit{Promise}\ m\ a$ and let $f :: a \to \mathit{Promise}\ m\ a$. By induction hypothesis, we have $p \equiv \mathit{Promise}\ (\mathit{fork}\ m)$ for some $m :: m\ a$ and $f \equiv \mathit{Promise} \circ \mathit{fork} \circ g$ for some $g :: a \to m\ b$. It follows that

$$\begin{aligned}
& p \ggg\!= f \\
&= \mathit{Promise}\ (\mathit{bindRef}\ (\mathit{fork}\ m)\ (\mathit{fork} \circ g)) \\
&= \mathit{Promise}\ (\mathit{fork}\ m \ggg\!= \lambda r \to \mathit{fork}\ (\mathit{read}\ r \geqslant \mathit{fork} \circ g \ggg\!= \mathit{read}) \\
&\equiv \mathit{Promise}\ (\mathit{fork}\ (m \ggg\!= \mathit{fork} \circ g \ggg\!= \mathit{read})) && \mathit{by}\ (10) \\
&\equiv \mathit{Promise}\ (\mathit{fork}\ (m \ggg\!= g)) && \mathit{by}\ (8)
\end{aligned}$$

which concludes the proof.

1.5. All laws in one diagram. The commutation of the following diagram is (?) equivalent with laws (8)–(10).

$$\begin{tikzcd}
 & & \mathit{Ref}_m\ a \arrow[d, "\mathit{return}"'] \arrow[drr, "\mathit{read}"] & & \\
a \arrow[d, "f"'] & m\ a \arrow[r, "\mathit{fork}"] \arrow[d, "\mathit{fmap}\ f"] & m\ (\mathit{Ref}_m\ a) \arrow[r, "\mathit{fmap}\ \mathit{read}"] & m\ (m\ a) \arrow[r, "\mathit{join}"] \arrow[d, "\mathit{fmap}\ (\mathit{fmap}\ f)"'] & m\ a \arrow[d, "\mathit{fmap}\ f"'] \\
b & m\ b \arrow[r, "\mathit{fork}"'] & m\ (\mathit{Ref}_m\ b) \arrow[r, "\mathit{fmap}\ \mathit{read}"'] & m\ (m\ b) \arrow[r, "\mathit{join}"'] & m\ b
\end{tikzcd}$$

with $\mathit{join}\ mm = mm \ggg\!= \mathit{id}$.

It implies $\mathit{join} \circ (\mathit{fmap}\ \mathit{read}) \circ \mathit{fork} \equiv \mathit{id}$ which is (8).

It implies $\mathit{fork} \circ \mathit{join} \circ (\mathit{fmap}\ \mathit{read}) \equiv \mathit{id}$ from which with deduce (8) (taking join).