An Equational Modeling of Asynchronous Concurrent Programming

David Janin(✉)

Univ. Bordeaux, CNRS, Bordeaux INP, LaBRI, UMR 5800, Bordeaux, 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 deadlock free. In this paper, we develop, study and extend 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 proving the soundness of the proposed equations and the applicability of the resulting API.

1 Introduction

Asynchronous Programming with Promises. Asynchronous programming is quite a popular approach for programming lightly concurrent applications such as, for instance, web services or, as shown recently, realtime signal processing and control [5]. 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 [12], asynchronous concurrent programming is nowadays available in most modern programing languages, including modern typed functional languages such as OCaml [9] and Haskell [8].

One of the reasons of such a success is that asynchronous programing is both comfortable and safe. Comfort comes from asynchronism, safety comes from deadlock freedom. 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.

A. Byrski and J. Hughes (Eds.): TFP 2020, LNCS 12222, pp. 180–203, 2020.
https://doi.org/10.1007/978-3-030-57761-2_9

Monads of Promises vs Promises of Monads. Quite interestingly, in most libraries, promises are defined with some *flavor* of a monad. Discussions about the true monadic nature of promises are numerous on the web[1], with various and contradictory conclusions depending on the considered host languages and libraries. Most of the resulting APIs even seem incompatible one with the others.

For instance, in a language like OCaml, where there is only an *implicit* IO monad, the monadic flavor of promises is made *explicit*. In OCaml *async* libraries [9], binding the fullfillement of a promise with some callback function is offered via an explicit *bind* function, and simple promises can be created by an explicit *return* function. In other words, OCaml promises are presented as if they form a monad.

On the contrary, in Haskell, where there is an *explicit* IO monad, there is no specific monad of promises [8]. Instead, the *async* library *extends* the IO monad to asynchronous concurrency with function $async :: IO\ a \to IO\ (Async\ a)$ that allows for forking (in a non blocking way) a IO action together with function $wait :: Async\ a \to IO\ a$ that allows for waiting and reading (in a non destructive way) the value returned by a forked action. Elements of $Async\ a$ can be seen as promises: promises of a returned value. However, while $Async$ has a functor instance, it does not have any monad instance. Haskell promises, when simply derived from Haskell *async* library, are not monadic.

Instead, aiming at relating both Haskell and OCaml *async* APIs, one can define another kind of promises in Haskell: elements of type $IO\ (Async\ a)$. Such an idea makes a lot of sense. An adequate function *return* can simply be defined by:

$$returnAsync :: a \to IO\ (Async\ a)$$
$$returnAsync\ a = async\ (return\ a)$$

However, as we shall see, there is only one possible associated *bind* function, defined by:

$$bindAsync :: IO\ (Async\ a) \to (a \to IO\ (Async\ b)) \to IO\ (Async\ b)$$
$$bindAsync\ m\ f = m \gg\!= \lambda r \to async\ (wait\ r \gg\!= f \gg\!= wait)$$

that *almost* yields a valid monad instance. For such an instance to be valid, we need to *restrict* further to those elements of $Async\ (m\ a)$ that are, up to equivalence, of the form $async\ m$ for some monadic action m (see Theorem 4). Then, the resulting monad of promises is shown to be *isomorphic* to the IO monad itself (see Theorem 5).

In other words, the possibility of a specific monad of promises in Haskell is a bit lost in the surrounding IO monad even though it can eventually be recovered.

Main Results. In this paper we shall first prove the above claims by understanding the properties satisfied by these asynchronous concurrent primitives, not only as specific instances of the *async* Haskell library, but from some more

[1] See e.g. Why are promises monads on stack overflow.

general properties they satisfy as elements of an asynchronous concurrent programing interface. For such a purpose, we shall define a fairly general notion of an asynchronous concurrent *extension* of an *arbitrary* monad, the current Haskell *async* library providing such an extension instance for the IO Monad.

One type function generalizing *Async* and two primitives generalizing *async* and *wait* are the key features of our proposal, defined as a Haskell type class. The expected semantics of these primitives is then specified by means of equational laws relating the behavior of the new primitives with the behavior of *return* and *bind* in the underlying monad. One may wonder why bothering capturing asynchronous concurrent semantics by means of equational laws. Elements of an answer are numerous and essentially the same as when asking why defining monad API semantics by means of monad laws.

These equations allow for defining unambiguously the semantics of the proposed asynchronous primitives which *properties* can therefore be examined in full details. We also seek at finding a smallest possible set of such primitives therefore increasing the *safety* of our proposal: only a small kernel of primitives needs to be implemented and proved correct, other needed asynchronous functions uniformly deriving from these primitives. The *correctness* of any instance of these primitives can also be checked against these equations, either by means of some derived test suites [1] or by formal proofs. Last, many of the proposed equations yield rewriting rules that can be used, at compile time, for *code optimization*, reducing the number of forked processes.

Such a general approach also opens the way for the programmer to design his or her own monad, a kind of a *domain specific monad*, with its specialized and safe API, and a specific asynchronous concurrent extension attuned towards the expected application. As an example, we have recently defined the notion of *timed monads* which asynchronous extensions are specific to the timed setting [6]. Indeed, in a timed monad, a timed action not only returns an explicit value but also an implicit duration. Promises associated to timed monad actions should also handle these durations.

The proposed formal approach eventually provides a better understanding of what *asynchronous concurrency* is, compared to *general concurrency*. Simply said, both are defined by means of some notions of *processes* that can be forked and, some corresponding (shall we say derived) notion of communication *channels* through which forked processes can communicate one with the other. However, a major difference lays in the way these communication channel are handled.

In asynchronous concurrency, communication channels are only created when forking a process, as (one way) broadcast channels from that forked processes. The resulting communication graphs are acyclic. On the contrary, in general concurrency, (two way) communication channels are freely created and passed as parameter of forked processes. The resulting communication graphs are arbitrary ones. The safety of asynchronous concurrency compared to general concurrency follows from this simple but crucial difference.

In this paper, such a difference is also be understood as a general guiding principle for developing asynchronous concurrency itself, beyond the simplest kind of processes one can define out of monad actions. As an example, we define (monadic) streams of values as inductively nested monad actions and we show that promises of such streams can simply be defined as streams of promises, that is, inductively nested monad promises. An associated artefact has been implemented in Haskell and available on the net[2]. It has been successfully applied to realtime synchronous processing and asynchronous control of audio streams [5] therefore illustrating the application scope of the proposed model.

Overview of Paper Content. Our presentation is organized as follows. After a short review of the basic features of functors and monads in Sect. 2, our abstraction of the *async* library is presented, in Sect. 3. There, a generic type class *MonadRef* specifies the asynchronous concurrent extension of a monad m, defined by an abstract type $Ref_m\ a$, whose elements, called *monad references* are created by $fork :: m\ a \to m\ (Ref_m\ a)$ and used by $read :: Ref_m\ a \to m\ a$. Simply said, every monad reference uniquely refers to a forked monad action which returned value can be read via that monad reference. 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*.

In Sect. 3, a first series of laws aim at capturing the basic semantics of monad references: how monad references are indeed bound to forked actions. These laws essentially states that *fork* and *read* behave in a coherent way with respect to the underlying monad law. This allows for proving in Sect. 4 that the type function $m \circ Ref_m$ is a functor and, under some adequate restrictions, also a monad isomorphic to m itself.

In Sect. 5, a second series of laws is more concerned with the asynchronous and concurrent nature of monad references. One idempotency and two commutation rules are stated for ensuring that actions are indeed executed when forked and read actions essentially have no side effects but waiting for the referenced actions to terminate. As an illustration of this 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 these additional rules.

In Sect. 6, we show how the notion of monad references can be lifted to more complex data types such as monadic streams. In some sense, monad streams can be forked into promises of monad streams that are simply encoded as streams of promises. Asynchronous concurrency is then further developed in Sect. 7. Defining a general notion of monad structure references, we eventually prove by examples that asynchronous concurrency is a fairly general programing paradigm that can be extended far beyond the existing libraries.

The general coherence and relevance of our proposal is illustrated throughout by defining a simple valid extension of Haskell IO monad, a self-contained simplified version of the existing Haskell *async* library.

[2] See https://github.com/djanin/TimedMonadStream.

Observational Equivalence in Haskell. Although aiming at achieving a formal modeling of asynchronous concurrency, throughout the paper most concepts are presented by means of Haskell type classes which instances are thus requested to satisfy some number of equational laws. Compared to a purely theoretical approach, such a presentation comes with some overhead. However, an immediate benefit is that it is directly applicable as demonstrated by associated libraries developed both in Haskell and, to a lesser extent, OCaml (see footnote 2).

Throughout the paper, we consider that two elements $a_1\ a_2 :: a$ of a given type a are (observationally) *equal* when there are *indistinguishable* in any context of use. In other words, denoting by $\equiv$ such an observational equality, we have $a_1 \equiv a_2$ when for any function $f :: a \to IO\ ()$, there is no observable difference between running $f\ a_1$ and $f\ a_2$ in the (idealized) IO monad. For instance, when instance of the class *Eq* with a defined equality $==$, the observational equivalence $\equiv$ in a type a is generally finer than the defined equality. Whenever $a_1 \equiv a_2$ we have $a_1 == a_2$. Indeed, the context function $\lambda x \to return\ (a_1 == x)$ distinguishes a_1 and a_2 in the case $a_1 == a_2$ is false[3]. In other words, observational equivalence in a given type depends on the primitives defined on that type.

2 Preliminaries on Monadic Functors and Monad Actions

For our presentation to be reasonably self-contained, we review below the definition and some properties of functors and monads, following the programmer's point of view offered by the pioneering works of Moggi [10] and Wadler [13].

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:

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

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

for every monad action $m :: m\ a$ and functions $f : a \to b$ and $g : b \to c$. In other words, the function *fmap* extends to typed functions the function m over types.

[3] This is a bit over simplified for one could also require such a distinguishing context to be itself definable in Haskell.

2.2 Monads

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

class *Functor* $m \Rightarrow$ *Monad* m **where**
$return :: a \to m\ a$
$(\gg\!=) :: m\ a \to (a \to m\ b) \to m\ b$

The infix operator $(\gg\!=)$ is called *bind* when used as a function. Elements of type $(m\ a)$ are called *monad actions*.

Every instance of the *Monad* class shall satisfies the following laws:

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

$$m \gg\!= return \equiv m \tag{4}$$

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

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

The first and second equations state that, in some sense, *return* acts 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.

2.3 Coherence Property

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

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

for every monad action $m :: m\ a$ and function $f :: a \to b$. This equation states that the mapping function induced by the monad primitives equals the mapping function defined in the parent *Functor* class instance. Indeed, one can check that if m is a monad then the function $\lambda f\ m \to m \gg\!= (return \circ f)$ satisfies both functor laws, i.e. any monad is indeed a functor.

2.4 Alternative Syntax for Binds

Haskell *do-notation* allows writing simpler composition of monad actions. Indeed, we may write:

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

the variables $x_1, x_2, \ldots, x_{n-1}$ possibly omitted when not used, in place for the bind series $m_1 \gg\!= \lambda x_1 \to m_2 \gg\!= \lambda x_2 \to ...m_{n-1} \gg\!= \lambda x_{n-1} \to m_n$ with m_1,

$m_2, \ldots, m_n$ 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 actions is necessarily evaluated from left to right. However, 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 $\mathit{getChar} :: \mathit{IO}\ \mathit{Char}$ and $\mathit{putChar} :: \mathit{Char} \to \mathit{IO}\ ()$ that respectively allows for getting the next character typed on the keyboard ($\mathit{getChar}$), or printing on the screen the character passed as argument ($\mathit{putChar}$). As a usage example, one can define the function:

$$\mathit{echo} :: \mathit{IO}\ ()$$
$$\mathit{echo} = \mathit{getChar} \gg\!= \mathit{putChar} \gg\!= \mathit{echo}$$

that, when ran, repeatedly waits for a character to be typed on the standard input and prints it out on the standard output.

An important feature of monadic IO actions, as monad actions, is that they are *not* executed unless passed to the top level. This illustrates the fact that, especially in a concurrent setting, monads, with *return* and *bind* functions, can be used for dynamically *defining* actions that can later be *run* or even *forked*.

Another important aspect of the IO monad in Haskell is that it is a *strict* monad in the sense that, when executing $\mathit{bind}\ m\ f$, the monadic action m is executed for its argument to be given to function f *before* evaluating f. This contrasts significantly with Haskell principle of lazy evaluation but clearly allows a better control, or even any control at all, on IO scheduling.

In other words, asynchronous extension of the IO monad offers a way to reintroduce (controlled) laziness and, therefore, parallelism into such a strictness.

3 Elementary Monad References

We describe here the first half of our formalization of promises by defining the notion of monad references with a first series of equations that suffices for analyzing the monadic nature of (the derived notion of) promises in Sect. 4. Analyzing the concurrent nature of these monad references is postponed to Sect. 5.

3.1 Monad Reference

Simply said, a monad reference is a reference to a "running" monad action, created by forking such an action. Such a monad reference can then be read for accessing the value returned by that referenced action. In terms of Haskell type classes, this yields:

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

where:

(a) $\mathit{Ref}_m\ a$ is a type of *monad references* bound to running actions of type $m\ a$,
(b) $\mathit{fork}\ m$ is an action that *launches* the execution of the monadic action m and (immediately) return a *monad reference* bound to that action,
(c) $\mathit{read}\ r$ is an action that (possibly) *waits for* and *returns* the *value* returned by the running action bound by the monad reference r,

respectively generalizing *Async a*, *async* and *wait* in the *async* library defined over Haskell IO monad.

3.2 Basic Semantics Laws

Every instance of the *MonadRef* class must first satisfy the following laws:

$$(\mathit{fork}\ m) \gg\!= \mathit{read} \equiv m \tag{7}$$

$$\mathit{fork} \circ \mathit{read} \equiv \mathit{return} \tag{8}$$

$$\mathit{fork}\ (m \gg\!= f) \equiv (\mathit{fork}\ m) \gg\!= \lambda r \to \mathit{fork}\ (\mathit{read}\ r \gg\!= f) \tag{9}$$

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

Intuitively, Law (7) states the basic semantics of forks and reads: reading a just forked action essentially behaves like that executing action, side effects included! Law (8) states that forking a read essentially amounts to returning an equivalent reference. In other words, reads followed by forks essentially behaves like kind of identities.

Last, Law (9) states that forking a bind can be decomposed into two successive forks, provided the reference returned by the first fork is passed through as argument of the second one. To some extent, binds distribute over forks.

In other words, these three laws essentially ensure that *fork* and *read* behave in a way compatible with the structure of the monad m. This will be formally stated in Sect. 4.

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 [7], is described by the following instance:

$$
\begin{array}{l}
\textbf{newtype}\ \mathit{MRef}\ a = \mathit{MRef}\ (\mathit{MVar}\ a) \\
\textbf{instance}\ \mathit{MonadRef}\ (\mathit{IO})\ \textbf{where} \\
\quad \textbf{type}\ \mathit{Ref}_{\mathit{IO}} = \mathit{MRef} \\
\quad \mathit{fork}\ m = \textbf{do}\ \{ v \leftarrow \mathit{newEmptyMVar}; \mathit{forkIO}\ (m \gg\!= \mathit{putMVar}\ v); \\
\qquad\qquad\quad \mathit{return}\ (\mathit{MRef}\ v)\} \\
\quad \mathit{read}\ (\mathit{MRef}\ v) = \mathit{readMVar}\ v
\end{array}
$$

With this definition, one can review all expected rules and check to which extent they are satisfied. Law (7) 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 (8) is less obviously satisfied. Indeed, two *distinct* mutable variables are created and we need them to be observationally equivalent, at least when used as monad references therefore encapsulated under *MRef*. It occurs that, they refer to two forked monad actions that both return 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 differences.

Law (9) validity essentially follows from the same reason, *forkIO* being non blocking and *readMVar* returning, in a non destructive way, the expected value essentially as soon as it is available.

Again, encapsulating mutable variables with *MRef* is crucial for hiding all the other primitives usually defined over *MVar*s such as, for instance, *takeMVar* that performs a destructive read on a mutable variable.

3.4 Comparison with *async* Library

Our definition of monad references is inspired and looks like a generalization of the *async* library of Haskell. As such, one could define instead another default IO instance of monad references by taking $\mathit{Ref}_{\mathit{IO}} = \mathit{Async}$, $\mathit{fork} = \mathit{async}$ and $\mathit{read} = \mathit{wait}$. Would such an instance be valid? It occurs that $\mathit{Async}\ a$ just as $\mathit{MVar}\ a$ is an instance of the class *Eq* therefore, as discussed in the introduction, this seems to prevent Laws (8) and (9) to be satisfied. However, encapsulating *Async*s just as done above for *MVar*s also solves such an issue. Then, one could prefer such a *Async* based instance for it offers a better support for interruption handling [8].

3.5 Counter Examples for Other Possible Laws

We have stated some equational properties that should 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 is our main source of counter-examples for examining other possible laws or bad instances. The reason for this is that the IO monad conveys 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 use several times in the text. With $m_0 :: m\ (Ref_{IO}\ Char)$ defined by $m_0 = getChar \gg\!= (fork \circ return)$, we have

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

even though 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 $fork\ (m_0 \gg\!= read)$ returns a similar reference since, by (7) and (4), it is equivalent to $fork\ (getChar)$. However, it is non blocking since *fork* is non blocking.

In other words, it is false that $fork\ (m \gg\!= read) \equiv m$ in general. However, in the next section, we shall use the fact that, provided $m = fork\ m'$ for some $m' :: m\ a$, then such an equation does hold.

4 Elementary Properties of Monad References

In this section, we study the properties deriving from our equational definition of elementary monad references. Readers more interested in using asynchronous extensions of a monad may directly jump to Sect. 5 for a discussion about the concurrent nature of monad references.

In this section, we 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 (7)–(9) for monad references primitives. As a matter of fact, the categorical property we examine here only depends on the above laws.

4.1 Induced Functor

Observe that, although $Ref_m :: * \to *$ is a type function, it cannot be a functor since there is no function that allows for creating or reading 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, one can prove:

Theorem 1. *The type function $m \circ Ref_m$ equipped with fmapRef defined by*

$$fmapRef :: MonaRef\ m \Rightarrow (a \to b) \to m\ (Ref_m\ a) \to m\ (Ref_m\ b)$$
$$fmapRef\ f\ m = m \gg\!= \lambda r \to fork\ (read\ r \gg\!= (return \circ f))$$

yields a valid functor instance.

Proof (sketch of). The fact that *fmapRef* satisfies law (1) follows from (8) and standard monad laws. The fact that *fmapRef* satisfies law (2) follows from (9) and standard monad laws. □

In other words, we have proved 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. The reader may find surprising that law (7), though describing the basic semantics of forks and reads, is not mentioned here. It turns out that it has 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 Sect. 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 *monad transformations*:

$$Fork :: m \dot{\to} m \circ Ref_m$$
$$Fork_a = fork :: m\ a \to m\ (Ref_m\ a)$$
$$Read :: m \circ Ref_m \dot{\to} m$$
$$Read_a = \lambda m \to m \gg\!= read :: m\ (Ref_m\ a) \to m\ a$$

defined for every type a, where $\dot{\to}$ denotes the (non Haskell) natural transformation type constructor, and we have:

Theorem 2. *Both Fork and Read are natural transformation, that is, for every function $f :: a \to b$ we have:*

$$fmapRef\ f\ (fork\ m) \equiv fork\ (fmap\ f\ m) \tag{10}$$

for every action $m :: m\ a$, and we have:

$$fmap\ f\ (m \gg\!= read) \equiv fmapRef\ f\ m \gg\!= read \tag{11}$$

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.*

Proof (sketch of). Equation (10) follows from Law (9), and Eq. (11) follows from Law (7). The fact $Read \circ Fork \equiv Id$ follows from Law (7). □

The reverse composition $Fork \circ Read$ is not the identity as shown in the IO monad by $fork\ (m \gg\!= read) \not\equiv m$ whenever m is a a blocking monad action. In other words, there may be more behaviors definable with monad references than behaviors definable without.

4.3 The Possibility of a Monad

We have shown that $m \circ Ref_m$ is a functor. Is this functor monadic? Strictly speaking, this is not true as we shall see here by enumerating all possible definitions for returns and binds.

First, up to equivalent definitions, the unique possibility of a function *return* is defined by:

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

for it is the unique uniformly defined function inhabiting its type. For a function *bind* there are four possible candidates:

$$bindRef :: m\ (Ref_m\ a) \to (a \to m\ (Ref_m\ b)) \to m\ (Ref_m\ b)$$

(a) $bindRef\ m\ f = fork\ (m \gg\!= read \gg\!= f \gg\!= read)$
(b) $bindRef\ m\ f = m \gg\!= \lambda r \to fork\ (read\ r \gg\!= f \gg\!= read)$
(c) $bindRef\ m\ f = m \gg\!= read \gg\!= \lambda a \to fork\ (f\ a \gg\!= read)$
(d) $bindRef\ m\ f = m \gg\!= read \gg\!= f$

Of course, the fact that such a list is, up to equivalence, complete necessitates a proof. One can observe that we have at least enumerated all possible insertions of a fork into the possible series of binds. In some sense, the IO monad instance forces to respect functional dependencies in sequence. Then it seems that adding additional forks and reads would essentially yield equivalent bind candidates thanks to rules (7)–(9).

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 candidate (b), we have:

$$fmapRef\ f\ m \equiv bindRef\ m\ (returnRef \circ f) \tag{12}$$

for every $m :: m\ (Ref_m\ a)$ *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 functions of the form* $fork \circ f$ *some* $f :: a \to m\ b$*, then the bind candidate (b) also satisfies law (3) in arbitrary monad instances.*

In other words, Lemma 3 states that the bind candidate (b) is a good candidate for us to prove that $m \circ Ref_m$ is our expected monad of promises *provided* we restrict ourselves to the subtype of $m\ (Ref_m\ a)$ defined by elements of the form $fork \circ m$ for some monad action $m :: m\ a$.

4.4 The Expected Monad of Promises

In Haskell, such an expected subset[4] of $m\ (Ref_m\ a)$ is defined by the type:

$$\textbf{newtype}\ Promise\ m\ a = Promise\ \{\, thePromise :: m\ (Ref_m\ a)\}$$

only equipped with the two primitives:

$$\begin{aligned} &forkP :: MonadRef\ m \Rightarrow m\ a \to Promise\ m\ a \\ &forkP = Promise \circ fork \\ &readP :: MonadRef\ m \Rightarrow Promise\ m\ a \to m\ a \\ &readP\ p = (thePromise\ p) \gg\!= read \end{aligned}$$

Theorem 4. *The following definitions are valid instances of the Functor and Monad type classes:*

$$\begin{aligned} &\textbf{instance}\ MonadRef\ m \Rightarrow Functor\ (Promise\ m)\ \textbf{where} \\ &\quad fmap\ f\ (Promise\ m) = Promise\ (fmapRef\ f\ m) \end{aligned}$$

and

$$\begin{aligned} &\textbf{instance}\ MonadRef\ m \Rightarrow Monad\ (Promise\ m)\ \textbf{where} \\ &\quad return = Promise \circ returnRef \\ &\quad (\gg\!=)\ (Promise\ m)\ f \\ &\qquad = Promise\ (bindRef\ m\ (thePromise \circ f)) \end{aligned}$$

with bind candidate (b) for bindRef.

Proof (sketch of). The fact *Promise m* is a functor follows from Theorem 1. The fact it is also a monad follows from Lemma 3 proving additionally, by induction on the complexity of their definition, that every definable inhabitant of *Promise m a*, that is, defined only with *ForkP*, *fmap*, *return* and *bind*, is equivalent with an element of the form *Promise* (*fork m*) for some $m :: m\ a$. □

Theorem 5. *Categorical functors m and Promise m are isomorphic.*

Proof (sketch of). Follows from Theorem 2 and the proof argument of Theorem 4, since, restricted to monad action of the form *fork m* with $m :: m\ a$, we indeed have $fork\ (fork\ m \gg\!= read) \equiv fork\ m$, by law (7), therefore *Promise* ∘ *Fork* is the inverse of *Read* ∘ *thePromise*. □

As a special case within the explicit Haskell IO monad, we thus have defined a monad of promises quite smilar to the one defined in the implicit IO monad of OCaml [9].

[4] We could call it a subtype for it has fewer inhabitants. However, since it also supports fewer operations, such a name for type *Promise m a* would be confusing.

Of course, the purpose of such a definition of promises in Haskell is merely for stating and proving the above theorems. Programmers *are not* advised to use such a definition of promises for this would result in having to combine two distinct monads. Instead, from now on, we simply use monad references and the associated primitives, therefore staying within the underlying monad m, much in the same way one would use the *async* library staying within the IO monad.

5 Concurrent Monad References

We aim at capturing asynchronous concurrent behaviors by means of the notion of monad references. However, as we shall soon see, Laws (7)–(9) fail to achieve by themselves such a goal.

5.1 Pathological Instances

Each of the following instances, though satisfying Laws (7)–(9), violates (at least) one of the intuitive properties we expect asynchronous concurrent primitives to satisfy.

Read Effect-Freedom (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.

$$
\begin{aligned}
&\textbf{instance}\ \mathit{MonadRef}\ \mathit{IO}\ \textbf{where}\\
&\quad \textbf{type}\ \mathit{Ref}_{\mathit{IO}} = \mathit{IO}\\
&\quad \mathit{fork} = \mathit{return} :: \mathit{IO}\ a \to \mathit{IO}\ (\mathit{IO}\ a)\\
&\quad \mathit{read} = \mathit{id} :: \mathit{IO}\ a \to \mathit{IO}\ a
\end{aligned}
$$

Such an instance, that could be generalized to an arbitrary monad, is valid. Indeed, law (7) follows from (3), law (8) is immediate, and law (9) follows from (5). However, reading such a kind of reference just amounts to performing the referenced action. Read actions therefore have *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 provides a counter example to that claim.

$$
\begin{aligned}
&\textbf{instance}\ \mathit{MonadRef}\ \mathit{IO}\ \textbf{where}\\
&\quad \textbf{type}\ \mathit{Ref}_{\mathit{IO}} = \mathit{MRef}\\
&\quad \mathit{fork}\ m = m \gg\!= (\mathit{MRef} \circ \mathit{newMVar})\\
&\quad \mathit{read}\ (\mathit{MRef}\ v) = \mathit{readMVar}\ v
\end{aligned}
$$

where $\mathit{newMVar} :: a \to m\ (\mathit{MVar}\ a)$ creates a new mutable variable filled with its argument. Compared to the instance of IO references given in Sect. 3, we just have changed the definition of *fork*. In this new instance, forking an action waits for that action to be completed before returning a reference. As a consequence, with $m = \mathit{getChar}$, the action $\mathit{fork}\ m$ is now blocking.

Read Independence (C). With a bit more of coding, the following instance, although with non blocking forks, violates our requirement that forking an action amounts to executing it.

```
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 }
```

In this instance, a forked action is indeed executed only when its associated reference is read for the first time.

5.2 Concurrency Laws

We aim now at designing a second series of equational laws, called concurrency laws, that enforce the properties detailled above. From now on, these additional laws must also be satisfied by any monad instance of the class *MonadRef*.

Following a typical approach of concurrency theory, these additional laws simply state that certain idempotency and commutation properties are satisfied:

$$read\ r \equiv read\ r \gg read\ r \tag{13}$$

$$\begin{aligned} fork\ m_1 \gg\!= \lambda r_1 \to (fork\ m_2 \gg\!= \lambda r_2 \to return\ (r_1, r_2)) \\ \equiv fork\ m_2 \gg\!= \lambda r_2 \to (fork\ m_1 \gg\!= \lambda r_1 \to return\ (r_1, r_2)) \end{aligned} \tag{14}$$

$$\begin{aligned} read\ r_1 \gg\!= \lambda x_1 \to (read\ r_2 \gg\!= \lambda x_2 \to return\ (x_1, x_2)) \\ \equiv read\ r_2 \gg\!= \lambda x_2 \to (read\ r_1 \gg\!= \lambda x_1 \to return\ (x_1, x_2)) \end{aligned} \tag{15}$$

for every monad reference $r\ r_1\ r_2 :: Ref_m\ a$ and monad action $m_1\ m_2 :: m\ a$, with $m_1 \gg m_2$ denoting the composition $m_1 \gg\!= \lambda\_ \to m_2$.

5.3 Discussion on Concurrency Laws

The intuitive meaning of these laws is detailed below.

By stating that reading actions are idempotent, Law (13) implies that values returned by read actions only depends on their parameter reference, i.e. reads are non destructive, and that side effects associated to readings a given reference occurs during the execution of the first (terminated) occurrence of such a read. After the return of a first read, any further reading of the its reference is side effect free and essentially instantaneous. The pathological instance (A) is ruled out by such a law. Indeed, there, monad references are arbitrary monad actions and reading amounts to executing them. Still, both pathological instances (B) and (C) satisfy such a law.

By stating that fork actions commute, Law (14) enforces 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*. With instance (B) the forks of these two actions will surely not commute. However, the pathological instance (C) still satisfy such a law.

Last, by stating that read actions commute, Law (15) enforces the fact that the execution of a forked actions cannot depend on the associated readings. Monad references must truly refer to *running* monad actions, and read action cannot have other side effect but waiting for the these running action to terminate. 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. The resulting reads do not commute.

In other words, these three 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.

5.4 Validity in the Asynchronous Concurrent Extension of the IO Monad

In the IO instance of monad references defined in Sect. 3, law (13) follows from the fact that the action *readMVar* is non destructive.

Law (14) is perhaps the most debatable one. It may wrongly suggest that the side effects of action m_1 and m_2 commute. This is not true. In the concurrent framework of Haskell, these side effects are executed in parallel therefore, up to the possible non determinism induced by that parallelism, forking m_1 right before m_2 or forking m_2 right before m_1 essentially produces the same side effects.

Law (15) is easily accepted as valid since reads essentially wait for termination of (parallel) forked actions. Waiting for the termination of one action and then another just amounts to waiting for the termination of both.

5.5 Commutation Rules and Induced Non Determinism

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:

$$fork\ (putChar\ \texttt{'a'}) \gg fork\ (putChar\ \texttt{'b'})$$
$$fork\ (putChar\ \texttt{'b'}) \gg fork\ (putChar\ \texttt{'a'})$$

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

$$fork\ (getChar) \gg\!= \lambda r_1 \rightarrow fork\ (getChar) \gg\!= \lambda r_2 \rightarrow read\ r_1 \gg\!= printChar$$
$$fork\ (getChar) \gg\!= \lambda r_2 \rightarrow fork\ (getChar) \gg\!= \lambda r_1 \rightarrow read\ r_1 \gg\!= printChar$$

that both non deterministically print either `'a'` or `'b'` when reading the string `"ab"` from the standard input.

6 Asynchronous Concurrency and Data Flow Programing

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. Even though this can easily be generalized to more complex structure, we simply review here the case of monadic streams as they can be used for data flow programing in Haskell [5].

6.1 Monad Streams

Monad streams are defined by the following inductive data type:

$$\textbf{data}\ \mathit{Stream}\ m\ a = \mathit{Stream}\ \{\mathit{next} :: m\ (\mathit{Maybe}\ (a, \mathit{Stream}\ m\ a))\}$$

In other words, a monad stream is essentially defined as a monad action that either returns nothing when the stream terminates, or just a value and the action defining the continuation of that stream otherwise. As an example of a monad stream, there is the standard input stream defined by:

$$\begin{aligned}&\mathit{stdinStream} :: \mathit{Stream}\ \mathit{IO}\ \mathit{Char}\\ &\mathit{stdinStream} = \mathit{Stream}\ \$\ \textbf{do}\ \{a \leftarrow \mathit{getChar}; \mathit{return}\ \$\ \mathit{Just}\ (a, \mathit{stdinStream})\}\end{aligned}$$

that, when executed, eventually returns all the characters typed from the standard input (*stdin*) one after the other.

A function printing a stream of characters to the standard output (*stdout*) can also be defined by:

$$\begin{aligned}&\mathit{streamStdout} :: \mathit{Stream}\ \mathit{IO}\ \mathit{Char} \rightarrow \mathit{IO}\ ()\\ &\mathit{streamStdout}\ (\mathit{Stream}\ m) = \textbf{do}\ \{c \leftarrow m; \textbf{case}\ c\ \textbf{of}\\ &\quad \{\mathit{Nothing} \rightarrow \mathit{return}\ (); \mathit{Just}\ (a, s) \rightarrow \textbf{do}\ \{\mathit{putChar}\ a; \mathit{streamStdout}\ s\}\}\}\end{aligned}$$

Then, the function *echo* described above as an example of function in the IO monad can then be recoded by:

$$\mathit{echo} = \mathit{streamStdout}\ \mathit{stdinStream}$$

Such an example illustrates fairly well the power of monad streams for data flow programing, a kind of monad programing technique fairly popular among Haskell programmers.

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 (or isochronous) 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 function that concatenates two monad streams one after the other.

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 instance for monad streams (called vertical) and the monad instance it induces.

6.4 Monad Stream References

Observe that *sharing* a monad stream such as *stdinStream* among several processes would result in *distributing* the standard inputs among these processes. The notion of monad references can be extended to monad streams and allows for *duplicating* monad streams. More precisely, there is a generalized notion of references applicable to monad streams defined by:

```
type StreamRef_m = Stream Ref_m
```

i.e. a reference to a monad stream is simply a stream of nested monad references.

Then, forking a monad stream and reading the resulting monad stream reference can simply be defined by:

```
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 effects. Such a possibility is especially useful in reactive on-the-fly data flow programming [5]. More formally, one can prove that:

Lemma 6. *For every* $s :: Stream\ m$ *we have:*

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

In other words, with monadic stream references defined as above, the first two laws (7)–(8) lift to the case of monadic stream references. For Eq. (9) to be satisfied by streams and stream references, we eventually need to equip monad streams with an adequate monad structure.

7 More Asynchronous Concurrency

In order to equip monad streams with an adequate monad instance, we eventually define additional (asynchronous) concurrent primitives that cannot be derived from the *read* and *fork* primitives defined so far.

7.1 More Concurrent Primitives

These primitives are specified by the following type class refinement of the type class *MonadRef*:

```
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:

(a) *tryRead* r is the action that immediately returns nothing if the referenced action is not terminated or just its returned value otherwise,
(b) *parReadRef* $r_1\ r_2$ is the action that returns the value of the earliest terminated referenced actions or, in the case both actions are already terminated or are terminating at the same time, either of the returned values.

In the IO monad, such additional monad reference primitives 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 }
```

We may aim at axiomatizing the behavior of these newly introduced primitives. For instance, one may expect to have:

$fork\ m \gg\!= tryRead \equiv m \gg\!= return$ when m is instantaneous,
$fork\ m \gg\!= tryRead \equiv return\ Nothing$ 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, a typing that is not (yet) available.

7.2 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 ← forkStream s1; r2 ← forkStream s2; return (next $ mergeRef r1 r2) }
```

with

```
mergeRef :: MonadRefPlus m ⇒
  Stream Ref_m a → Stream Ref_m a → Stream m a
mergeRef (Stream r1) (Stream r2) = Stream $ do
  { c ← parRead r1 r2; case c of {
      Left Nothing → next $ readT (Stream r2);
      Right Nothing → next $ readT (Stream r1);
      Left (Just (a, src1)) → return $ Just (a, mergeRef src1 sr2);
      Right (Just (a, src2)) → return $ Just (a, mergeRef sr1 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.

7.3 Derived Stream Monad

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

$$
\begin{array}{l}
\textbf{instance}\ \mathit{MonadRef}\ m \Rightarrow \mathit{Monad}\ (\mathit{Stream}\ m)\ \textbf{where} \\
\quad \mathit{return}\ a = (\mathit{Stream} \circ \mathit{return} \circ \mathit{Just})\ (a, \mathit{mempty}) \\
\quad (\gg\!=)\ (\mathit{Stream}\ m)\ f = \mathit{Stream}\ \$\ \textbf{do} \\
\qquad \{\, c \leftarrow m; \textbf{case}\ c\ \textbf{of} \\
\qquad\quad \{\, \mathit{Nothing} \rightarrow \mathit{return}\ \mathit{Nothing}; \\
\qquad\qquad \mathit{Just}\ (a, mc) \rightarrow \mathit{next}\ \$\ \mathit{merge}\ (f\ a)\ (mc \gg\!= f)\,\}\,\}
\end{array}
$$

There, the flattening operation essentially amounts to merge monadic substreams from the moment they appear. This feature is especially useful when handling asynchronous control flows [5].

Lemma 7. *For every stream* $s :: \mathit{Stream}\ m\ a$ *and function* $f :: a \rightarrow \mathit{Stream}\ m\ b$*, we have:*

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

In other words, the monad reference Law (9) also lifts to monad stream references.

7.4 Stream Monad as a Monad Extension

The above monad instance of $\mathit{Stream}\ m$ is also an extension of the monad m in the sense that, with:

$$
\begin{array}{l}
\mathit{liftStream} :: m\ a \rightarrow \mathit{Stream}\ m\ a \\
\mathit{liftStream}\ m = \mathit{Stream}\ \$\ \textbf{do}\ \{\, a \leftarrow m; \mathit{return}\ \$\ \mathit{Just}\ (a, \mathit{emptyStream})\,\}
\end{array}
$$

we have:

Lemma 8. *Function liftStream is a natural embedding of* m *into* $\mathit{Stream}\ m$ *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* $f :: a \rightarrow m\ b$*.*

7.5 Generalization Monad References to Monadic Structures

The above treatment of monadic streams seems to fit a fairly general notion of references to monadic structures. More precisely, we can define the type class:

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

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

$$(\mathit{forkT}\ s) \gg\!= \mathit{readT} \equiv \mathit{return}\ s \tag{16}$$

$$forkT \circ readT \equiv return \tag{17}$$

$$forkT\ (s \gg\!= f) \equiv (forkT\ s) \gg\!= \lambda r \rightarrow forkT\ (readT\ r \gg\!= f) \tag{18}$$

for every monad structure $s :: t\ m\ a$ and function $f :: a \rightarrow t\ m\ b$.

Then, thanks to Lemmas 6 and 7 there is the following valid instance for monad stream references:

instance $MonadRefPlus\ m \Rightarrow MonadDataRef\ Stream\ m$ **where**
$forkT = forkStream$
$readT = readStream$

It is probably the case that such a construction can be generalized to arbitrary monadic versions of inductive types. However, such a study goes out of the scope of the present paper.

7.6 More Parallelism

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

$forkAll :: (Traversable\ t, MonadRef\ m) \Rightarrow t\ (m\ a) \rightarrow m\ (t\ (Ref_m\ a))$
$forkAll = mapM\ fork$

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

$sortRefs :: (Traversable\ t, MonadRefPlus\ m) \Rightarrow t\ (Ref_m\ a) \rightarrow Stream\ m\ a$
$sortRefs = foldMap\ (liftStream \circ read)$

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, using functions *forkAll* and *sortRefs*, much like using primitives *fork* and *read*, is safe for it yields no deadlock.

7.7 Asynchronous vs General Concurrency

The above generic definition of *sortRef* 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 (fully) concurrent Haskell, this is not a necessity as shown by the following direct implementation of *sortRefs* in the IO monad:

$$
\begin{array}{l}
\mathit{sortRefs}_{IO} :: \mathit{Traversable}\ t \Rightarrow t\ (\mathit{Ref}_{IO}\ a) \to \mathit{IO}\ (\mathit{Stream}\ \mathit{IO}\ a) \\
\mathit{sortRefs}_{IO}\ t = \mathbf{do}\ \{ v \leftarrow \mathit{newEmptyMVar}; \\
\quad \mathit{mapM\_}\ (\lambda r \to \mathit{forkIO}\ (\mathit{read}\ r \gg\!= \mathit{putMVar}\ v))\ t; \\
\quad \mathit{return}\ \$\ \mathit{mvarToStream}\ v\ (\mathit{length}\ t)\} \\
\mathbf{where} \\
\quad \mathit{mvarToStream}\ \_\ 0 = \mathit{mempty} \\
\quad \mathit{mvarToStream}\ v\ n = \mathit{Stream}\ \$\ \mathbf{do} \\
\quad\quad \{ a \leftarrow \mathit{takeMVar}\ v; \mathit{return}\ \$\ \mathit{Just}\ (a, \mathit{mvarToStream}\ v\ (n-1))\}
\end{array}
$$

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 many programming possibilities as a more general concurrent programming framework. This is no surprise. This is the price to pay for the increase of robustness and safety offered by asynchronous concurrency compared to general concurrency.

8 Related Works and Conclusion

The study proposed here started as an attempt to clarify the properties of an existing and somewhat ad hoc but succesfull experiment of realtime audio processing and control in Haskell [5]. As such, it was first designed as a stand alone approach that was a priori not much related with former theoretical investigations. A posteriori, our proposal offers an equational formalization of the semantics of (a kernel of) the existing *async* library. To the best of our knowledge, no such an axiomatization has yet been proposed.

We present a fairly generic notion of a monad extension. A first series of laws describes how to go back and forth between the underlying monad m and its extension $m \circ \mathit{Ref}_m$ via a retraction pair of natural transformations. The underlying general category theoretic schema seems rather orthogonal to more classical existing techniques for combining monads [2,10]. Such a notion of a monad extension is probably worth being studied more in the depth.

A second series of laws enforces concurrency as shown by ruling out pathological instances. However, there is no guarantee our proposal is complete. There could well be other pathological instances violating our intuition on what asynchronous concurrency should be. Moreover, all our examples are based on extending the strict IO monad. The underlying intuition is somewhat biased. What is the asynchronous concurrent extension of a non strict monad is yet not that clear. The successful extensions of the notion of monad references to more complex structures, such as monad streams or traversable structures of monad actions, only constitute partial answers to that question.

As already observed, the possibility of defining an equational theory for the additional primitives such as *tryRead* bumped into the lack of a denotional semantics that describes action durations. For instance, we cannot describe the property that between to monadic actions, one is finishing before the other unless

they both block endlessly. A pure denotational approach might still be possible following the recent proposal of a timed extension of Scott domains [4]. However, investigating such a possibility goes out of the scope of the present paper.

Last, one could also examine the possibility of defining asynchronous concurrency as an algebraic effect [11]. The termination of a forked action indeed sounds like raising an effect that is eventually passed to all the readers of the monad reference bound to that action. The resulting API would be different than the current one. How the proposed axiomatization could be adapted to such a distinct modeling approach is an open problem.

References

1. Claessen, K., Hugues, J.: QuickCheck: a lightweight tool for random testing of Haskell programs. In: International Conference on Functional Programming (ICFP) (2000)
2. Dahlqvist, F., Parlant, L., Silva, A.: Layer by layer - combining monads. In: Theoretical Aspects of Computing (ICTAC), pp. 153–172 (2018)
3. Halstead, R.H.: Multilisp: a language for concurrent symbolic computation. ACM Trans. Program. Lang. Syst. **7**(4), 501–538 (1985)
4. Janin, D.: Spatio-temporal domains: an overview. In: Fischer, B., Uustalu, T. (eds.) ICTAC 2018. LNCS, vol. 11187, pp. 231–251. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-02508-3_13
5. Janin, D.: Screaming in the IO monad. In: ACM Workshop on Functional Art, Music, Modeling and Design (FARM). ACM Press (2019)
6. Janin, D.: A timed IO monad. In: Komendantskaya, E., Liu, Y.A. (eds.) PADL 2020. LNCS, vol. 12007, pp. 131–147. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-39197-3_9
7. Peyton Jones, S., Gordon, A., Finne, S.: Concurrent Haskell. In: Principles of Programing Languages (POPL). ACM, New York (1996)
8. Marlow, S.: Parallel and Concurrent Programming in Haskell. O'Reilly, Sebastopol (2013)
9. Minsky, Y., Hickey, J., Madhavapeddy, A.: Real World OCaml: Functional Programming for the Masses. O'Reilly, Sebastopol (2013)
10. Moggi, E.: A modular approach to denotational semantics. In: Pitt, D.H., Curien, P.-L., Abramsky, S., Pitts, A.M., Poigné, A., Rydeheard, D.E. (eds.) CTCS 1991. LNCS, vol. 530, pp. 138–139. Springer, Heidelberg (1991). https://doi.org/10.1007/BFb0013462
11. Pretnar, M.: An introduction to algebraic effects and handlers. In: The 31st Conference on the Mathematical Foundations of Programming Semantics (MFPS XXXI). Electronic Notes in Theoretical Computer Science, vol. 319, pp. 19–35 (2015)
12. Reppy, J.H.: Concurrent Programming in ML. Cambridge University Press, Cambridge (1999)
13. Wadler, P.: Comprehending monads. In: Conference on LISP and Functional Programming (LFP). ACM, New York (1990)