Ministry of Science and Higher Education of the Russian Federation
ITMO UNIVERSITY

GRADUATION THESIS

APPLICABILITY OF FREE(R) APPLICATIVES TO STRUCTURING CONCURRENT PROGRAMS

Author: Yurchenko Artem Olegovich

Subject area: 01.03.02 Applied mathematics and informatics

Degree level: Bachelor

Thesis supervisor: Aksenov V.E., PhD

Saint Petersburg, 2021

CONTENTS

INTRODUCTION

Free monads, as often defined, are an abstraction that given a functor provides a monad, i.e. given a functor you get a monad "for free"[13, Chapter 6].

Freer (or operational) monads are, in a sense, even more free: they don't require a functor. Given a type of kind `* -> *` they yield a monad[6]. It is convenient to get rid of the functor constraint, so we will use this "kind" of free monads.

Free monads are widely used in Haskell to create embedded domain specific languages (eDSLs). A developer supplies a data type that describes basic operations of the language and gets a monadic interface to interact with these operations[1]. The interface can later be interpreted and operations executed.

However, free monads, as usually defined, are inherently sequential. Any interpreter can only execute one request at a time.

Free(r) applicatives are similar to free(r) monads except that they return applicative functors instead of monads[1]. The effect that it has is that while expressive power of applicative functors is lesser than that of monads, their structure allows for a more profound static analysis. Interpreters of free(r) applicatives have access to (and can execute) several operations at a time.

What that means is that now we can build interpreters that can execute operations in parallel, potentially speeding up programs written with free applicatives. Unfortunately, a lot of programming logic cannot be expressed via applicative functors alone.

So, let us combine the two concepts and use monadic features when the expressive power is needed and applicative features when it is possible. Thus we regain the ability to program elaborate logic while having access to many operations simultaneously.

It matters, because "the free lunch is over", has been for at least 15 years. It is time to do concurrency and concurrent programs are hard to write[12].

With the chosen approach, we can once write a suitable interpreter for a language, write a suitable executor for operations and enjoy “free” parallelization for all programs in that language.

The trick is that just executing all operations available to the interpreter at the time is not enough. Operations might have side-effects. Thus, results of a sequential execution might differ drastically from results of a parallel execution.

The goal of this research was to implement a free monad/applicative structure with an interpreter that allows for semantically correct parallel execution of operations. We put accent on accommodating side-effecting operations, which has not been done before, to our knowledge.

As a proof of concept, we implemented a parallelizable state transformer language and benchmarked its performance with different interpreters and executors.

The code for the work can be found at `https://gitlab.com/grepcake/playground`.

This document is structured the following way. The first chapter provides a more in-depth overview of the basic concepts: free(r) monads, applicatives, their usage, applicative relation to concurrency and existing approaches that exploit this relation. The first chapter also outlines the issue of side-effecting operations, which were out of scope of the existing utilities. Finally, it lists specific goals of this project.

The second chapter describes in detail a new free monad data structure and explains motivation for the new design. It also explains the chosen concurrency model (Mazurkiewicz traces) and concurrency abstractions that aid algorithms (independency relation and dependence graphs). Speaking of algorithms, the second chapter presents an algorithm to run FreeP over side-effect free operations and two algorithms to run FreeP over side-effecting operations making use of the two abstractions.

The third chapter talks about implementation details. It introduces *runners* to run languages considering concurrency abstractions defined on the languages; and *multi-executors* that controls how exactly operations are executed in parallel (or sequentially). The third chapter also mentions a paral-

lel state-transformer language designed as a proof-of-concept of the implementation correctness and worth.

CHAPTER 1. OVERVIEW OF RELATED TOPICS

1.1. Free(r) monads as an approach to creating eDSLs

Let us introduce a freer monad data type and its Monad instance in Listing 1 as described in “Freer Monads, More Extensible Effects”[6]. The Functor and Applicative instances are induced by the Monad instance.

Listing 1 – The freer monad

```
data FFree f a where
  Pure :: a -> FFree f a
  Impure :: f x -> (x -> FFree f a) -> FFree f a

instance Monad (FFree f) where
  return a = Pure a

  Pure x >>= f = f x
  Impure fx cont >>= f = Impure fx (\x -> cont x >>= f)
```

From here on, we will call the type a “free monad” for convenience. Basically, the free monad above is a sequence of `Impure` steps that finish with the `Pure` value. `Impure` steps additionally allow the following steps to consume its value, which is reflected in the `(x -> FFree f a)` argument.

Let us show an example of how the free monad could be used in creating eDSLs in Listing 2.

Listing 2 – A free monad DSL

```
data Operation a where
  GetInt :: Operation Int
  PrintString :: String -> Operation ()

type OpM = FFree Operation

getInt :: OpM Int
getInt = Impure GetInt Pure

printString :: String -> OpM ()
printString s = Impure (PrintString s) Pure

greetANumber :: OpM ()
greetANumber = do
  printString "Enter a number to greet"
  x <- getInt
  printString ("Hello, number " <> show x)
```

A developer creates a type of basic operations Operation, it has two constructors representing two operations. There is an operation that takes no arguments and returns an integer number and then an operation that takes a string to print and returns a unit. A unit is a nullary tuple `()`, which basically bears no information and often serves to represent "returning nothing" in Haskell.
A bit lower in the listing the developer creates a type synonym for their new language OpM. It is FFree Operation and is a monad.
For convenience, we "lift" the basic operations from the operation type to the monad in functions getInt and printString.
Finally, we make a little program in the language that makes use of the monadic interface. It requests a number and then prints a greeting to that number.

However, notice that we haven't specified behaviour for any of the operations. We have only defined their structure: their arguments, their return types and nothing more. There is no way to run the program right now. Helpfully, for a freer monad over f it holds that given a transformation from f to a monad m, there is a monad homomorphism from FFree f to m. A function to illustrate the principle is shown in Listing 3.

Listing 3 – foldFFree

```
foldFFree :: Monad m => (forall x. f x -> m x) -> FFree f a -> m a
foldFFree exec ffree = case ffree of
  Pure x -> pure x
  Impure fx cont -> exec fx >>= \x -> foldFFree exec (cont x)
```

In practice, it means we can only write an executor for the operation type and get an executor for the free monad over it for free. IO is a standard Haskell type for working with input/output, mutability etc in Haskell. It is natural to write an executor into this type. It is displayed in Listing 4.

Mind you, we could have supplied a different operations executor. A pure executor (for example, one that always gives a predetermined number) could be useful for testing.

1.2. Applicatives and parallelism

The free monad structure in Listing 1 is inherently sequential. Its interpreter, one like that on Listing 3, has access only to one Impure step. The

Listing 4 – Free executor

```
exec :: Operation a -> IO a
exec GetInt = fmap read getLine
exec (PrintString s) = putStrLn s

run :: FFree Operation a -> IO a
run = foldFFree exec

> run greetANumber
Enter a number to greet
5
Hello, number 5
```

following steps are hidden behind functions of type `(x -> FFree f a)`. In order to access them, we have to execute the current operation to get that `x`.

On the other hand there is another Free structure that does not have such a limitation. It is free applicatives. One of many possible definitions of a free applicative is shown in Listing 5.

Listing 5 – A free applicative

```
data FreeAp f a where
  PureA :: a -> FreeAp f a
  ImpureA :: f x -> FreeAp f (x -> a) -> FreeAp f a

instance Functor (FreeAp f) where
  fmap f (PureA x) = PureA (f x)
  fmap f (ImpureA fx next) = ImpureA fx (fmap (f .) next)

instance Applicative (FreeAp f) where
  pure = PureA

  (<*>) :: FreeAp f (a -> b) -> FreeAp f a -> FreeAp f b
  PureA f <*> rhs = fmap f rhs
  ImpureA fx next <*> rhs = ImpureA fx (flip <$> next <*> rhs)
```

This construction is very similar to that of the free monad. It is a sequence of `ImpureA` steps. However, the resulting value (of type `x`) of a step operation cannot take part in construction of the following steps. It is represented by the fact that the following step, signified by `FreeAp f (x -> a)`, does not have `x` as a function parameter, unlike the monadic `(x -> FreeAp f a)`. Expressive power of the applicative is lesser than that of the monad.
On the other hand, there is an important benefit of that construction. All following steps are available at once, they are not hidden behind a func-

tion. That means that we can statically examine steps and possibly execute several of them in parallel.

1.3. Combining free applicatives and monads to aid concurrency

So far, we have free applicatives that allow to examine several operations at once and free monads that allow to use results of preceding operations in following operations.
It raises a question whether a compromise is possible. Is it possible to design a data structure that would allow a Monad instance *and* examination of several operations at once, when they don't depend on each other's results? It appears, that, yes, it is.

First of all, there is a natural way to express computations that base their construction upon results of previous computations. It is computations built via a monad function `>>=`. Its signature is on Listing 6.

Listing 6 – Monadic bind
```
(>>=) :: Monad m => m a -> (a -> m b) -> m b
```

It "chains" together two computations. The second computation is built based on the value of the first computation.
At the same time, there is also a natural way to express computations that are constructed independently of each other. It is computations built via an applicative function `<*>`. Its signature is on Listing 7.

Listing 7 – Applicative <*>
```
(<*>) :: Applicative f => f (a -> b) -> f a -> f b
```

However, since every Monad is an Applicative, `<*>` can be expressed in terms of a Monad. See `ap` in Listing 8.

Listing 8 – Monad ap
```
ap :: Monad m => m (a -> b) -> m a -> m b
ap mf mx = mf >>= (\f -> mx >>= (\x -> return (f x)))
```

In fact, it is a law that `<*>` should be equal to `ap`[5]. If we follow the law to the letter, however, it is obvious that we cannot benefit from the inherent

parallelism of Applicatives: all operations should be strictly sequential in the order specified by ap. However, the *spirit* of the law is that the result of the computation should not differ whether it's using ap or <*>. <*> might have a different implementation and even execute both its arguments in parallel when it doesn't change the program's result.
This is the trick that is used by two tools of automatic parallelization: Haxl and fraxl.

In 2014 Simon Marlow et al. built a Haxl framework. In their paper[8] they describe a Fetch datatype. It is an ad hoc free monad. Its operations are requests for data from various data sources. Haxl supposes that all requests are side-effect free, so any two operations can be safely executed in parallel. Fetch, similarly to other free constructs, consists of a series of steps, each step having a batch of requests to fulfill. <*> implementation of Fetch batches together requests from its arguments. Fetch interpreter executes all requests of a step simultaneously.

Haxl is very practical, but it has some issues. It can only be run in IO, it has a baked-in inescapable caching, requires runtime-checked initialization and some other problems outlined by Will Fancher. In 2015 Will Fancher made a more general utility: fraxl[4]. It is an actual free monad. It has several benefits to it including more type safety, less "magic", generalization of the monad m that an eDSL can be interpreted into (it is always IO in the case of Haxl), handles a wider range of operations, provides a monad transformer etc. However, it still makes the same assumption that all operations are side-effect free.

1.4. The fraxl free monad

The Free Monad of Will Fancher[3] was an inspiration for the resulting data structure. The exact construction of our free monad will be discussed in the second chapter. For now, it is worth mentioning how Fancher's original free monad works. Fancher employs a "Free Monad given an Applicative". It is a free monad that given an applicative returns a monad. Its construction is given in Listing 9.
Here Applicative instance of Free basically offloads <*> implementation to the underlying parameter f.

Listing 9 – A free monad given an applicative

```
data Free f a where
  Pure :: a -> Free f a
  Free :: f (Free f a) -> Free f a

instance Functor f => Functor (Free f) where
  fmap f (Pure a) = Pure (f a)
  fmap f (Free a) = Free (fmap (fmap f) a)

instance Applicative f => Applicative (Free f) where
  pure = Pure
  Pure a <*> Pure b = Pure $ a b
  Pure a <*> Free mb = Free $ fmap a <$> mb
  Free ma <*> Pure b = Free $ fmap ($ b) <$> ma
  Free ma <*> Free mb = Free $ fmap (<*>) ma <*> mb
  -- ^ The important line

instance Applicative f => Monad (Free f) where
  Pure a >>= k = k a
  Free a >>= k = Free (fmap (>>= k) a)
```

<*> and ap are different in this implementation. The (<*>) = ap is not followed to the letter. Seemingly, it might lead to some issues and Fancher provides an example (shown in Listing 10).

Listing 10 – Const breaks the program(?)

```
newtype Const a b = Const a

instance Monoid m => Applicative (Const m) where
  pure _ = mempty
  Const f <*> Const v = Const (f `mappend` v)

liftFree (Const "a") <*>  liftFree (Const "b") = Free (Const "ab")
liftFree (Const "a") `ap` liftFree (Const "b") = Free (Const "a")
```

The results are clearly different, and it might seem like an issue. It is not. As Fancher goes on to explain that law is not literal, it should hold up to some interpretation. In case of free monads, it is up to their interpreter, e.g. foldFFree in Listing 3. So the trick is to disallow interpreting constructions similar to the described one with Const. It is solved by lifting requirements to the executor function in foldFree/foldFFree interpreters. It should be not just any function, but an applicative homomorphism. A function (morphism) f is an applicative homomorphism if it satisfies two conditions in Listing 11.

Listing 11 – Applicative homomorphism

```
f :: Applicative g => g a -> h a
f (pure x) = pure x
f (u <*> v) = f u <*> f v
```

However such free monad requires an applicative instance for the operation type. It is unfortunate, but `FreeAp`, defined earlier, provides an applicative for any operation type. So, the final and actual definition of the fraxl free monad is in Listing 12.

Listing 12 – The fraxl free monad

```
type Free' f = Free (FreeAp f)
```

1.5. Side effects

It is often the case, that operations might have side-effects, for example, “insert” operations of a database eDSL. Such languages are out of scope of both Haxl and fraxl. However, if we want to support such languages, it is not immediately clear what can or should be done. The operations type is opaque, its definition is unavailable to the free data structure. It requires some additional framework over the operations type to reason about what can be executed in parallel.

Thankfully, concurrency theory has many models and abstractions to choose from[2]. There are π- and ρ-calculi[10], Petri nets[15], Mazurkiewicz traces[9] etc.

1.6. Goals

With all that in mind, we’ve set up the following goals:

— Figure out a free monad/applicative structure suitable to analyze operations with side-effects;
— Find or devise a concurrency model to express the notion of “safety to execute in parallel”;
— Build a system making use of the devised concepts;
— Create some language via this system that was not possible in existing approaches, show benefits of parallelization.

Conclusions on Chapter 1

In the first chapter we briefly review free monads as a way to create embedded domain specific languages. One starts with defining a basic operations type that contains all operations that should be present in their eDSL. Then they apply a free monad to that type and get a monadic interface for their language for free. In order to execute programs in that language, we must write an interpreter from the free monad into some other monad. In general, it is enough to only write an executor for the operation type into that other monad. Due to free monad structure, it is enough to get an interpreter for the free monad applied to the operation type.

We also discuss how free applicatives aid in static analysis and parallelism. By their construction, free applicatives have access to all computational steps at once, unlike free monads.

This observation naturally leads to an idea of "mixing" a free monad construction with a free applicative construction to get benefits of both approaches to a certain extent. We show a theoretical way to combining the concepts. However, that way breaks a law on applicative-monad relationship if considered literally. On the other hand, if we follow the spirit of the law, it is enough for it to hold "up to interpretation".

Then we show two examples that use the "mixed" free monad/applicative. The first is the Haxl framework, which uses its Applicative instance to "batch" together computations of both `<*>` arguments. The second is fraxl, which is similar to Haxl but is represented by a general free monad. We mention a "Free Monad given an Applicative" employed by fraxl. It allows to guarantee the applicative-monad relationship law preservation if an operation executor is an applicative homomorphism.

Finally, given that both examples do not support side-effecting operations, we state our goal to design a system that can do that.

CHAPTER 2. NEW PARALLELIZING FREE MONAD STRUCTURE FOR EFFECTFUL OPERATIONS

2.1. A new free monad structure: FreeP

A suitable data structure in some sense has to support both monadic and applicative features. It needs to be able to build following computations based on results of previous operations. At the same time, computations that do not have such dependency should be statically accessible preferably all at once.

After some experiments we've come up with the type shown in Listing 13 (FreeAp is the same as in Listing 5).

Listing 13 – The FreeP free monad

```
type FreeP f = Free (FreeAp f)

data Free f a where
  Last :: f a -> Free f a
  Step :: f x -> (x -> Free f a) -> Free f a

instance Functor f => Functor (Free f) where
  fmap f (Last fa) = Last (fmap f fa)
  fmap f (Step fx next) = Step fx (fmap f . next)

instance Applicative f => Applicative (Free f) where
  pure x = Last (pure x)

  Last ff <*> Last fa = Last (ff <*> fa)
  Last ff <*> Step fx next = Step ((,) <$> ff <*> fx) (\(f, x) ->
    fmap f (next x))
  Step fx next <*> rhs = Step fx (\x -> next x <*> rhs)

instance Applicative f => Monad (Free f) where
  Last fa >>= rhs = Step fa rhs
  Step fx next >>= rhs = Step fx (\x -> next x >>= rhs)
```

This construction is very similar to the fraxl free monad (see Section 1.4). There is a "free monad given an applicative" (but slightly different), there is a FreeAp that provides an Applicative instance for any operation type and the actual free monad is a "composition" of the two.

There is one but important difference though. It lies in the "free monad given an applicative" <*> function. The fraxl free monad relies on

the fact that no two operations can influence each other. That means any two might be grouped together. So it tries to group (via <*>) operations as early as possible: `Free ma <*> Free mb = Free $ fmap (<*>)ma <*> mb`.

If both the LHS and the RHS are "steps" that hold some operations, that `Free` immediately "batches" them together. Thus, we abandon the "canonical", `ap`-like left-to-right sequencing of operations. Instead, operations are interleaved. We can't afford that. Our operations might have side-effects. That means that some operations must be executed in the canonical, left-to-right order. In fact, any operation at *any step* of the LHS might need to be executed before operations in the RHS. In order to make a judgment whether an operation can be executed before other, following operations, this operation needs to have access to them. However, free monads hide all following steps with other operations behind a function. So, we can only "batch" together operations with the *last* step of the monad. If it is the last one, there are no more uninspectable steps and operations.

There comes the second difference: the difference in the "free monad given an applicative" definition. The fraxl analog (Listing 9) does not have a way to designate a "last step". It is either *a* step with some following steps, or the `Pure` end of the way. Since we explicitly need a way to tell if a step is the last one, let's make a constructor `Last` that represents exactly that.

2.2. Side-effect free operations

First of all, let us show that this structure can already work side-effect free operations, like fraxl and Haxl. First, we present a `foldFree` interpreter for the `Free`. It receives an applicative homomorphism (for reasons specified in Section 1.4) and returns a monad homomorphism: basically a way to run the `Free` code in some other monad. The implementation is presented in Listing 14.

Listing 14 – foldFree

```
foldFree
  :: (Monad m, Applicative f)
  => (forall x. f x -> m x) -> Free f a -> m a
foldFree exec (Last fa) = exec fa
foldFree exec (Step fx next) =
  exec fx >>= (\x -> foldFree exec (next x))
```

It is not enough to interpret FreeP, which is a Free with the operation type wrapped in FreeAp. We also need to write a FreeAp interpreter that executes operations.
The actual parallelism of the system depends on the FreeAp interpreter. It might choose to execute all operations in parallel (they all are side-effect free), execute them in some order sequentially or somehow combine this methods. For example, we will show a FreeAp interpreter that executes operations in a left-to-right order, in Listing 15.

Listing 15 – runFreeAp

```
runFreeAp
  :: Applicative g
   => (forall x. f x -> g x) -> FreeAp f a -> g a
runFreeAp _ (PureA a) = pure a
runFreeAp exec (ImpureA fx next) =
  (\x f -> f x) <$> exec fx <*> runFreeAp exec next
```

Now, a side-effect free operations interpreter seems to be easy to come up with as shown in Listing 16:

Listing 16 – foldFreeP

```
foldFreeP :: Monad m => (forall x. f x -> m x) -> FreeP f a -> m a
foldFreeP exec = foldFree (runFreeAp exec)
```

However, it is only legal if runFreeAp exec is an applicative homomorphism. Thankfully, it is, as is proven in Appendix A. However, other FreeAp interpreters should also satisfy this condition.

2.3. Suitable concurrency model

As we've mentioned in Section 1.5, it is unclear how to reason about side-effects of opaque operations, they need some additional framework to work with. So, we could either come up with some model ourselves or try to use one of many existing ones. Finding an existing theory would be a better option: in that case we could make use of work and ideas of other, smart, people. However since there are so many of concurrency abstractions, finding a proper one was not an easy task. We've reviewed some material on π-calculus and Petri nets but they either had some undesirable

properties (e.g. non-determinism) or were too complex. That lead to us creating certain abstractions outside of any framework. Later though we were immensely helped by the "Independence Abstractions and Models of Concurrency" paper[2], which guided us to Mazurkiewicz traces[9]. In fact, all of our abstractions have fit very well into that theory, with only slight adjustments or no adjustments at all.

2.3.1. Trace monoids and independency relation

Trace theory operates on a set of elements. It is tempting to say that in our case this set of elements is all values of the operation type. However, the operation type is *higher-kinded*. It has a kind of $* \text{ -> } *$, which means that it is not a type (a set of values), but it takes a type as a parameter and then provides a type. So, in our case the set of elements is a union of all types provided by the operation type.

There should be a *dependency* relation D on that set. It is any finite, reflexive and symmetric relation. It is complemented by an *independency* relation I, that is symmetric and irreflexive. It might seem too restrictive for I to be irreflexive. Surely, if some operation is side-effect free it is natural for it to be *independent* from itself. However, trace theory in that regard is only concerned with operations *reordering*. Since it is the same operation, any two instances of it are indistinguishable and introducing an order on them does not make sense. However, parallel execution is different. It is a known issue that, for example, executing two increment operations at once could result in only one increment. A similar concept is a Write-after-Write data hazard[11, Page 335]. So, we relax the reflexivity requirement on a dependency relation and irreflexivity — on the independency relation.

The theory defines the *trace equivalence* for a dependency D as the least congruence $\equiv_D$ in the set of sequences of operations, such that for all a, b

$$(a, b) \in I_D \Rightarrow ab \equiv_D ba$$

where I_D is the independency relation induced by D

Equivalence classes of $\equiv_D$ are called *traces* over D. It is easy to view sequences in one trace as programs that execute operations in a different

order but still have the same result. Moreover, operations that commute can also be executed in parallel.

A monoid over traces is a *trace monoid* also called a *free partially commutative monoid*. The concatenation operation of this monoid directly correlates to <*> of FreeP. <*> also basically concatenates two partially commutative sequences of operations.

It is pretty simple to represent a *dependency* or *independency* relation in Haskell via a type class. Independency maps pretty well to the notion of two operations being safe to execute in parallel, so we introduced the Parallelizable Haskell type class representing independency, as shown in Listing 17.

Listing 17 – Parallelizable type class

```
class Parallelizable f where
  independent :: f a -> f b -> Bool
```

The type class has one law: symmetricity (Listing 18).

Listing 18 – Parallelizable symmetricity

```
a `independent` b = b `independent` a
```

The algorithm to interpret FreeP with side-effecting operations that have an independency relation specified on them is similar to foldFreeP (Listing 16). It does not require modification of foldFree, but we need to adjust runFreeAp to take into account that some operations do not commute. The devised algorithm takes the biggest prefix of operations that *do* commute, executes them, supplies their results to the suffix and repeats until the suffix is empty. The actual implementation in Haskell requires a somewhat complex type-level machinery and is described in more detail in the third chapter.
Unfortunately, this algorithm is quadratic in number of consequent independent operations.

2.3.2. Dependence graphs

In trace theory, trace monoids are isomorphic to dependence graphs.

Dependence graphs over dependency D are Directed Acyclic Graphs where each vertex is labeled with some operation. Moreover, labels of two vertices are in the dependency relation if and only if these vertices are connected by an arc or are the same vertex.

We give the arcs the following semantics: if there is an arc from vertex u to vertex v, it means that u can not be executed in parallel with or after executing v.

The practical utility of dependence graphs is that we can ask a user to supply us a dependence graph of operations or some transitive reduction of it. Building such a graph might have a complexity better than quadratic one. Then we employ a $O(V + E)$ algorithm to execute operations, where V is the number of vertices and E is the number of arcs. The algorithm to break vertices into pools of independent operations is shown in Listing 19. After pooling we just execute operations pool by pool and return their results in the correct order.

Listing 19 – Dependence Graph Pooling

```
1:  function PoolGraph(N, C)
                                          ▷ N is an array of graph vertices
                                ▷ C is an array of nodes' children, basically an adjacency array
2:      D ← zero-filled array of size |N|      ▷ counters of yet unpooled parents
3:      for all n ∈ N do
4:          for all c ∈ C[n] do
5:              D[c] += 1

6:      P ← {n ∈ N. D[n] = 0}                                  ▷ the first pool
7:      R ← []                                                 ▷ list of resulting pools
8:      while P ≠ ∅ do
9:          R ← R ++ [P]                                       ▷ add the new pool to the result
10:         P' ← ∅                                             ▷ the next pool
11:         for all n ∈ P do
12:             for all c ∈ D[n] do
13:                 D[c] −= 1
14:                 if D[c] = 0 then          ▷ the node has no unsatisfied dependencies
15:                     P' ← P' ∪ {c}
16:         P ← P'
17:     return R
18: end function
```

Let's prove that this algorithm "works". It means proving three things:

1 correctness: all dependent operations will end up in the later pool.

2 optimality: all operations end up in the earliest possible pool.

3 computational efficiency: computational complexity is acceptable
Correctness is obvious. While at least one of vertex n's parents is not in any pool, $D[n] > 0$ by the algorithm construction. Hence, it will not get into the same pool as its parents.

Let's prove optimality. Suppose that it is not satisfied and there is a node n that breaks the condition with the smallest pool number i. Since it breaks the condition, it could have been placed in the pool j, $j < i$ (1) (that's why i can't be the first pool number). n could make it into the pool i only when some of its parents got into the pool $i - 1$ (by the algorithm construction). If n was placed in the pool j, in order to preserve correctness, the parents would have to be placed in the pool $j' \leqslant j - 1 < i - 1$. That means that we've found nodes with an even lesser pool number that break the condition. It is a contradiction.

As for computational efficiency, the algorithm takes $O(V + E)$ iterations: lines 2, 6, 7 and 17 take $O(V)$ iterations together and for loops on lines (3–5) and (8–16) each walk every arc once, so they take $O(E)$. Even if there was a way to make a faster algorithm, construction of the graph already takes $O(V + E)$ so it doesn't matter.

Conclusions on Chapter 2

In this chapter we describe a theoretical model of the desired system. We design a new free monad construction to accommodate effectful computations. Unlike the fraxl free monad it batches operations of the left argument of <*> with operations of the right argument only if the left operations are the last *step* of the left argument. The motivation is that operations of the right argument semantically *follow* operations of the left argument. However, if we were to put some right operations into a non-final step of the left argument, they would not be able to examine the following left operations. They would not be able to check whether there is a side-effecting left operations that *must* be executed before them and the semantic meaning of the program would be changed.

Then we mention the chosen concurrency model, that is *trace theory*, and abstractions that were found to be useful for the system. These abstractions are an independency relation and dependence graphs. An independency relation on operations signifies that two adjacent operations are safe

to re-order. We adjust the definition slightly to account for parallel execution. Dependence graphs represent graphs of operations to execute where arcs connect nodes that are dependent on each other and should be executed in the order implied by the arc direction. Finally we discuss algorithms that can make use of the said abstractions to devise a correct execution order of operations.

CHAPTER 3. IMPLEMENTATION, EXAMPLES AND BENCHMARKS

We chose Haskell as a language of implementation. It is sufficiently expressive to implement the model in a generally straightforward way. Also, Haskell is a language of implementation of the two predecessors: Haxl and fraxl.

3.1. FreeP

The FreeP implementation is completely straightforward and largely follows code listings in the second chapter.

Each one of FreeP interpreters utilizes foldFree from Listing 14. Everything else depends completely on a FreeAp interpreter supplied to foldFree . This interpreter has two responsibilities. First, it must order operations as dictated by our concurrency model. Second, it must execute separated batches of independent operations. These responsibilities are basically independent, so we separated them into two concepts: *runners* and *multi-executors*.

3.2. Multi-executors

A *multi-executor* is a function that takes a batch of independent, safe to execute in parallel operations, executes them and returns results.
Since operations might be parameterized by different types we use type-heterogeneous collections, i.e. collections where each element might be parameterized by a different type. In particular, we use extensible records provided by the `vinyl` package. The resulting type signature of a multi-executor is shown in Listing 20.

Listing 20 – A type signature of a multi-executor

```
Rec f rs -> m (Rec Identity rs)
```

f in that signature represents an operation type. rs is a type level list of types: rs = [r_1, r_2 ,.., r_n]. Rec f rs is a heterogeneous list of values: Rec f rs = [v_1 :: f r_1, v_2 :: f r_2, .., v_n :: f r_n].
The argument is a list of operations to execute. Multi-executors return a list in m of operation results Rec Identity rs. Identity is needed because Rec requires some *interpretation* over rs. Identity a is isomorphic to a, so it is not an issue.

The additional benefit of using `Rec` is that we make use of the Haskell type-checker to exclude certain programming mistakes. For example, it is impossible to accidentally omit some value from the resulting list. It would reflect on the type of the `Rec` and wouldn't type-check.

A multi-executor can do more than just execute. Let us imagine a database query language, that supports an "insert" operation. For most databases it is true that inserting multiple data in one query is more beneficial than doing multiple inserts. A multi-executor for this language might take all insert operations and perform them all as one insert. Such multi-executors are very language-specific, but it is possible to define several useful multi-executors applicable to every language. All of these executors receive an additional `forall a. f a -> m a` parameter. It is an executor of a single operation.

First, there is a plain sequential multi-executor that executes operations left-to-right one after another. It is called `sequential` and is shown in Listing 21.

Listing 21 – sequential

```
sequential
  :: Applicative m
  => (forall b. f b -> m b) -> Rec f rs -> m (Rec Identity rs)
sequential exec = rtraverse (fmap Identity . exec)
```

Then there is an `allConcurrent` multi-executor that executes each operation in a separate Haskell thread, shown in Listing 22.

Listing 22 – allConcurrent

```
allConcurrent
  :: MonadUnliftIO m
  => (forall b. f b -> m b) -> Rec f rs -> m (Rec Identity rs)
allConcurrent exec actions = withRunInIO $ \runInIO ->
  (NoPool.runConcurrently
    (rtraverse
      (NoPool.Concurrently . coerce . runInIO . exec) actions))
```

Note, that concurrency in Haskell is usually an IO prerogative. MonadUnliftIO[1] in Listing 22 is a way to extend that privilege to a slightly wider set of types, but we won't go into detail.

As lightweight as Haskell threads are it still might be expensive to spawn them on every operation. In that case an inTaskGroup multi-executor might be useful. As a first argument inTaskGroup takes a TaskGroup that allows to launch tasks in a fixed-size thread pool. The multi-executor is shown in Listing 23.

Listing 23 – inTaskGroup

```
inTaskGroup
  :: forall f m rs. MonadUnliftIO m
  => TaskGroup
  -> (forall b. f b -> m b)
  -> Rec f rs -> m (Rec Identity rs)
inTaskGroup tg exec =
  rtraverseAsync tg (fmap Identity . exec)

rtraverseAsync
  :: forall f h g rs. (MonadUnliftIO h)
  => TaskGroup -> (forall x. f x -> h (g x)) -> Rec f rs -> h (Rec
      g rs)
rtraverseAsync tg f rs = withRunInIO (`go` rs)
  where
    go :: (forall a. h a -> IO a) -> Rec f rs' -> IO (Rec g rs')
    go _ RNil = pure RNil
    go runInIO (x :& xs) =
      Pool.withAsync tg (runInIO (f x)) $ \ax -> do
        next <- runInIO (rtraverseAsync tg f xs)
        executedX <- Pool.wait ax
        pure (executedX :& next)
```

3.3. Runners

Runners are FreeP interpreters that offload the task of parallel execution to a multi-executor that they take as a parameter. There are three of them: runSideEffectFree, runParallelizable and runWithDependenceGraph.

runSideEffectFree is a runner for languages with side-effect free operations. That means that all operations are safe to execute in parallel. The runner turns FreeAp of every step into a Rec and feeds it to its multi-executor. It is shown in Listing 24

[1] https://hackage.haskell.org/package/unliftio-core-0.2.0.1/docs/Control-Monad-IO-Unlift.html#t:MonadUnliftIO

Listing 24 – runSideEffectFree

```
runSideEffectFree
  :: forall f m a. Monad m
  => (forall rs. Rec f rs -> m (Rec Identity rs))
  -> FreeP f a -> m a
runSideEffectFree multiExec = foldFree (go RNil)
  where
    go :: Rec f rs -> FreeAp f (Curried rs a') -> m a'
    go inds (ImpureA fx next) = go (fx :& inds) next
    go inds (PureA x) = runcurry' x <$> multiExec inds
```

runParallelizable is a runner that takes into account the independency relation (see Subsection 2.3.1) defined on the operation type. It acts according to the algorithm described in Subsection 2.3.1. The code is show in Listing 25.

Listing 25 – runParallelizable

```
runParallelizable
  :: forall f m a. (Parallelizable f, Monad m)
  => (forall rs. Rec f rs -> m (Rec Identity rs))
  -> FreeP f a -> m a
runParallelizable multiExec = foldFree (runParallelizableAp
   multiExec)

runParallelizableAp
  :: forall f m a. (Parallelizable f, Applicative m)
  => (forall rs. Rec f rs -> m (Rec Identity rs))
  -> FreeAp f a -> m a
runParallelizableAp multiExec = go RNil
  where
    go :: Rec f rs -> FreeAp f (Curried rs a') -> m a'
    go inds (ImpureA fx next)
      | rall (independent fx) inds = go (fx :& inds) next
    go inds next = flip runcurry' <$> multiExec inds <*> goOn next

    goOn :: FreeAp f a' -> m a'
    goOn (PureA x) = pure x
    goOn next = go RNil next

rall :: (forall x. f x -> Bool) -> Rec f rs -> Bool
rall _ RNil = True
rall cond (fx :& rest) = cond fx && rall cond rest
```

The final and the most intricate runner is a runWithDependenceGraph runner. The full code for it takes a lot of space, so we won't show it here

but it is available in the repository[2]. We will touch over the main function body without going into other functions used. See Listing 26.

Listing 26 – runWithDependenceGraph

```
runWithDependenceGraph
  :: forall f m a. Monad m
  => GraphT f m ()
  -> (EffArray f -> m (EffArray Identity))
  -> FreeP f a
  -> m a
runWithDependenceGraph buildGraph multiExec = foldFree (go RNil)
  where
    go
      :: (RecordToList rs, RMap rs)
      => Rec f rs -> FreeAp f (Curried rs a') -> m a'
    go effRec (ImpureA fx next) = go (fx :& effRec) next
    go effRec (PureA f) = do
      okayList <- executeEffRec effRec
      pure (unsafeApplyEffs effRec f okayList)

    executeEffRec
      :: (RecordToList rs, RMap rs)
      => Rec f rs -> m [Eff Identity]
    executeEffRec effRec = do
      let effArray = effsToEffArray effRec
      let work = interpretGraph buildGraph
      adjMap <- runGraphTImpl work effArray
      let pools = makePools (bounds effArray) adjMap
      results <- executePools pools effArray multiExec
      pure (Array.elems results)
```

The first argument of runWithDependenceGraph, buildGraph is an eDSL routine that builds a dependence graph (possibly, transitively reduced). The second argument is a multi-executor with a slightly modified type signature. Instead of a type-homogeneous Rec it operates on a simple array of Effs. These are two aspects worth discussing in more detail, because they are user-facing.

The eDSL in question is a Free monad over the GraphEff operation type, shown in Listing 27.

It has two operations: one to inspect a nodes array, where nodes mean vertices of the graph. The vertices are guaranteed to be ordered in the canonical, left-to-right order. Note that the canonical order is actually a topological

[2] https://gitlab.com/grepcake/playground/-/blob/375ed957fc51168d4e57ef680d46e4a43afb7b
src/GraphT.hs

Listing 27 – GraphEff

```
data GraphEff f a where
  InspectNodes
    :: (forall arr. IArray arr (Eff f)
    => arr Int (Eff f) -> a) -> GraphEff f a
  Connect :: Int -> Int -> a -> GraphEff f a
```

sorting of the resulting dependence graph. The other operation connects two vertices, specified by their indices in the array, with an arc. The graph should be acyclic, and indices grow in a topological order, so we always connect the lesser of the supplied indices with the greater one to avoid cycles. We also ignore loops, that are connections of an index with itself. First, they don't make sense, second, removing them might be considered a transitive reduction. Exposing this interface as an eDSL prevents users from seeing the internal structure of the graph or accidentally changing it in ways they are not supposed too.

The change in multi-executor type is also explained by the graph structure. If we were to label nodes' result types in their signatures, we would have to make signatures of all graph operations significantly more complex, e.g. type-level proofs of a node existence in the graph at the given index. Instead, we wrap all operations into an opaque `f` that only preserves the operation higher-kinded type but not the operation result type. Execution of wrapped operations hides result types as well, as is clear from the multi-executor signature. Wrapped results are then *coerced* into results of type expected by the program. It might seem unsafe. However, almost no interface is exposed to a user that would allow them to misuse `Eff`s. As long as the multi-executor *does* just execute the array operations, the underlying types do not change and it is safe to coerce them back. There are issues if the multi-executor returns an empty array, swaps cells in the resulting array, takes a subset of the resulting array etc. However, they would have to go out of their way to do that, it is not likely to be an accidental mistake.

The new multi-executor signature required writing a new set of multi-executors, but their implementation mostly follows that of the existing ones.

The rest of the runner is largely glue between the interface and the algorithm from Subsection 2.3.2.

3.4. Parallelizable state transformer

In order to test whether the implementation is correct we wrote a language that allows parallelizable state-transforming operations. The language is called PS for *P*arallelizable *State*-transformer. The operations are read and write operations on mutable references and arrays. The operation type for the language is shown in Listing 28.

Listing 28 – PSEff

```
data PSEff ref arr a where
  NewRef :: v -> PSEff ref arr (ref v)
  ReadRef :: ref v -> PSEff ref arr v
  WriteRef :: ref v -> v -> PSEff ref arr ()

  NewArray :: Int -> v -> PSEff ref arr (arr v)
  ReadArray :: arr v -> Int -> PSEff ref arr v
  WriteArray :: arr v -> Int -> v -> PSEff ref arr ()
  GetArrayLength :: arr v -> PSEff ref arr Int
```

In order to make use of the runParallelizable runner, the type needs a Parallelizable instance. It is pretty clear that a read and a write operation on the same reference do not commute, as well as two writes operations on the same references. In case of arrays, it is similar but the subject is determined by both the array and the index. The code is shown in Listing 29.
In that listing we also introduce two type classes to determine whether two references and two arrays are the same one.

We also provide an IO executor for the language. The obvious choice for the reference type is IORef. Unfortunately, we can't compare two IORefs with different parameter types. It would make sense to compare them as pointers, but IORef does not expose such interface.

The solution is to make a custom "pointer". Basically, every time we create a new reference or an array in our language, we assign it an atomically incrementing counter that serves as an identifier for that variable or that array. That requires carrying an IORef with that pointer. The standard way to express that is to use a ReaderT monad transformer. Thus the inter-

Listing 29 – Parallelizable instance for PSEff

```
class RefEq ref where
  refEq :: ref a -> ref b -> Bool

class ArrEq arr where
  arrEq :: arr a -> arr b -> Bool

instance (RefEq ref, ArrEq arr) =>
  Parallelizable (PSEff ref arr) where
  independent (ReadRef ref) (WriteRef ref' _) =
    not (refEq ref ref')
  independent (WriteRef ref _) (ReadRef ref') =
    not (refEq ref ref')
  independent (WriteRef ref _) (WriteRef ref' _) =
    not (refEq ref ref')
  independent (ReadArray arr i) (WriteArray arr' i' _) =
    not (arrEq arr arr' && i == i')
  independent w@WriteArray{} r@ReadArray{} =
    independent r w
  independent (WriteArray arr i _) (WriteArray arr' i' _) =
    not (arrEq arr arr' && i == i')
  independent _ _ = True
```

pretation monad evolves into PSIO, the reference — PSRef, the array — PSArr. All the types are defined in Listing 30.

The "phantom" parameter s in PSIO s a serves to protect the uniqueness of identifiers in all PSIO routines. It utilizes the same trick as ST[7]. In practice, it means that reusing references or arrays in different PSIO functions is not possible. The runPSIO function that executes a PSIO action in IO is in Listing 31.

The executor is fairly simple and mostly just "lifts" equivalent IO functions to PSIO. It can be seen in Listing 32.

The buildGraph routine required for the dependence graph runner is also pretty simple. It stores a hash map of last read and write accesses to references and array cells and draws arcs according to that information.

3.5. Matrix inverse

Despite having a PS language, it was hard to write anything meaningful to benchmark using only primitive PS operations. So we wrote a little matrix language on top of PS and a matrix inverse function using Gaussian

Listing 30 – PSIO and related types

```
type ID = Word64

data PSRef s a = PSRef
  { psIORef :: IORef a
  , psrID :: ID
  }

instance RefEq (PSRef s) where
  refEq (PSRef _ i) (PSRef _ j) = i == j

data PSArr s a = PSArr
  { psArr :: MutableArray RealWorld a
  , psaID :: ID
  }

instance ArrEq (PSArr s) where
  arrEq (PSArr _ i) (PSArr _ j) = i == j

type IDCnt = ID

newtype PSIO s a = PSIO (ReaderT (IORef IDCnt) IO a)
  deriving newtype (Functor, Applicative, Monad, MonadIO,
    MonadUnliftIO)
```

Listing 31 – runPSIO

```
-- don't want s to escape, it could lead to RefCnt reset
runPSIO :: (forall s. PSIO s a) -> IO a
runPSIO (PSIO psio) = do
  refCntRef <- newIORef 0
  runReaderT psio refCntRef
```

elimination. The reader can see the implementation in the repository[3]. The code is covered by tests that check all combinations of runners and multi-executors.

3.6. Benchmark results

We benchmarked the inverse operation on matrices of size 10×10 and 15×15. Since memory write/read operations are not usually expensive, the overhead of parallelization is more significant than speedup. So, we additionally slow down operations by 15 milliseconds. The reason for such a big slowdown is because it's done via threadDelay and the operating system

[3] https://gitlab.com/grepcake/playground/-/blob/375ed957fc51168d4e57ef680d46e4a43afb7b examples/ps/src/PS/Matrix.hs

Listing 32 – PSIO executor

```
type Exec = forall s a. PSIOEff s a -> PSIO s a

exec :: Exec
exec (NewRef v) = PSIO $ do
  idCntRef <- ask
  liftIO (PSRef <$> newIORef v <*> atomicGetAndInc idCntRef)
exec (ReadRef ref) = liftIO (readIORef (psIORef ref))
exec (WriteRef ref v) = liftIO (writeIORef (psIORef ref) v)
exec (NewArray n initial) = PSIO $ do
  idCntRef <- ask
  liftIO (PSArr <$> newArray n initial
                <*> atomicGetAndInc idCntRef)
exec (ReadArray arr i) = PSIO (liftIO (readArray (psArr arr) i))
exec (WriteArray arr i v) =
  PSIO (liftIO (writeArray (psArr arr) i v))
exec (GetArrayLength arr) = pure (sizeofMutableArray (psArr arr))
exec (Trace v) = PSIO (liftIO (putStrLn v))
```

timer is often not precise enough to use lesser delays. The results are displayed in Figure 1 and show certain speedup. Benchmarking was done via the criterion library[4] that provides statistically robust information on performance. A detailed report can be found in the repository[5]. It's clear from the report that confidence intervals of sequential multi-executors lie further on the time scale than confidence intervals of parallel multi-executors. Thus, we can observe some benefits to parallelization.

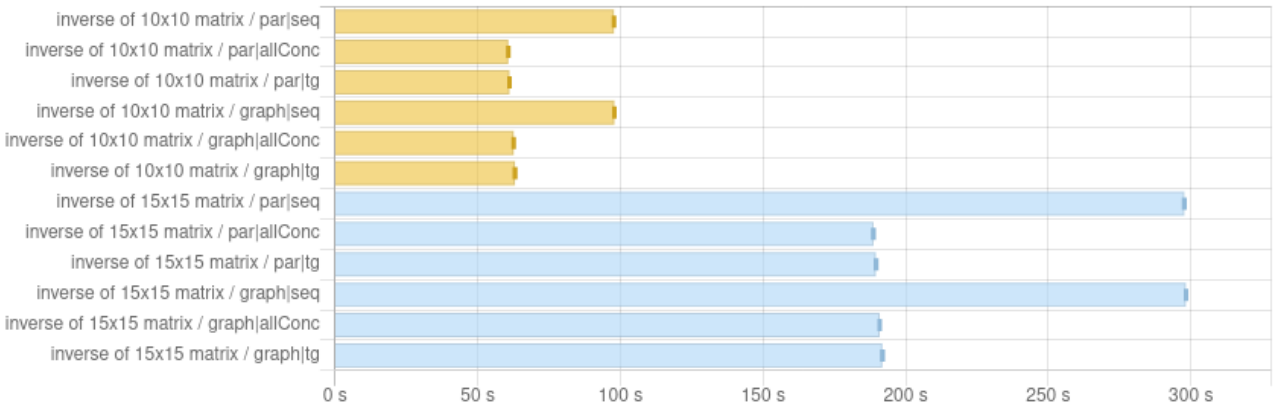

Figure 1 – Benchmarking results, 15 milliseconds/operation delay

[4] http://www.serpentine.com/criterion/

[5] https://gitlab.com/grepcake/playground/-/blob/2f302289779f8f07a2556e1179551b11d9dbb8
examples/ps/artifacts/benchmarking-report.html

3.7. Applicability advice

At last, we would like to give some advice on when the system would serve the best. First, basic operations should be relatively time-consuming. It is clear from the benchmark we used: we had to slow down each operation execution in order to overcome parallel execution overhead. A good example of "slow" operations are network requests or database manipulation. Second, the less "dependent" operations the program has the greater parallelization potential is. Each "dependent" operation, whether introduced by a monadic bind or via trace theory abstractions, makes the order "stricter" and reduces the amount of parallel operations in each batch fed to a multi-executor.

Conclusions on Chapter 3

In this chapter we go over the program implementation of the system. The free monad structure itself closely follows theoretical listing. We describe parallel execution utilities: multi-executors and runners.

Multi-executors are responsible for executing several operations in parallel. We give several examples, including a sequential multi-executor, a multi-executor that executes each operation in a separate Haskell thread and a multi-executor that executes operations in a thread pool.

Runners are responsible for providing a correct execution order that preserves semantic meaning of the program. They use the algorithms based on the concurrency model abstractions. These algorithms were described in the previous chapter.

Finally, we describe a proof-of-concept eDSL based on our free monad. It is a language with mutable variables and arrays. Furthermore, we build an algebraic operations language on top of that eDSL. We benchmark a program of that language. It is important that memory operations are usually fast and parallelization overhead does not make benefits clear. For that reason we slow down the operations executor by 15 milliseconds. We provide a graph representing benchmarking results that shows that parallel multi-executors indeed result in noticeable speedup.

CONCLUSION

This work presents a new free monad structure that allows to statically analyze operations and safely parallelize programs. It supports operations with side-effects unlike previous works. The work embeds operations into a trace theory model. It presents algorithms that provide a sufficient non-total order on operations that guarantees safe parallelization of non-ordered operations. The algorithms make use of trace theory abstractions.

We also provide a proof-of-concept system implemented in Haskell. It allows to create languages based on our free monad. The system features several language interpreter templates based on the devised algorithms. It also features several templates to execute a batch of independent operations.

We show a parallel state-transformer language “PS” built with the free monad and an algebraic operations language on top of “PS”. We provide a test-suite to ensure correctness of the implementation. We also benchmark the algebraic operations language and show benefits of parallelization.

We see further potential for exploration of this combination of a free monad a free partially commutative monoid and its place in category theory. The structure’s performance could further benefit from using Church-encoding and type-aligned sequences.

REFERENCES

1 Paolo Capriotti and Ambrus Kaposi. — “Free Applicative Functors”. — In: *Electronic Proceedings in Theoretical Computer Science* 153 (June 2014), p. 2–30. — ISSN: 2075-2180. — DOI: 10.4204/eptcs.153.2. — URL: %5Curl%7Bhttp://dx.doi.org/10.4204/EPTCS.153.2%7D.

2 Vijay D’Silva, Daniel Kroening, and Marcelo Sousa. — “Independence Abstractions and Models of Concurrency”. — In: *Verification, Model Checking, and Abstract Interpretation*. — Ed. by Ahmed Bouajjani and David Monniaux. — Cham: Springer International Publishing, 2017, — P. 151–168. — ISBN: 978-3-319-52234-0.

3 Will Fancer. — *More on Applicative Effects in Free Monads*. — https://web.archive.org/web/20210120121414/https://elvishjerricco.github.io/2016/04/13/more-on-applicative-effects-in-free-monads.html. — 2016.

4 Will Fancher. — *Fraxl: Better Concurrency and Caching for Free*. — https://github.com/ElvishJerricco/fraxl. — 2016.

5 Haskell Prelude contributors. — *Monad laws*. — https://web.archive.org/web/20210506155438/http://hackage.haskell.org/package/base-4.15.0.0/docs/Prelude.html. — [Online; accessed 18-May-2021]. — 2021.

6 Oleg Kiselyov and Hiromi Ishii. — “Freer Monads, More Extensible Effects”. — In: — Vol. 50. — Mar. 2015. — DOI: 10.1145/2887747.2804319.

7 John Launchbury and Simon Peyton Jones. — “Lazy Functional State Threads”. — In: *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* 29 (July 1998). — DOI: 10.1145/773473.178246.

8 Simon Marlow et al. — “There is No Fork: An Abstraction for Efficient, Concurrent, and Concise Data Access”. — In: *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*. — ICFP ’14. — Gothenburg, Sweden: Association for Computing Machinery, 2014, — P. 325–337. — ISBN: 9781450328739. — DOI: 10.1145/

2628136.2628144. — URL: %5Curl%7Bhttps://doi.org/10.1145/2628136.2628144%7D.

9 Antoni Mazurkiewicz. — “Introduction to Trace Theory”. — In: (Mar. 1995). — DOI: 10.1142/9789814261456_0001.

10 Robin Milner. — “The Polyadic π-Calculus: a Tutorial”. — In: (Sept. 2000). — DOI: 10.1007/978-3-642-58041-3_6.

11 David A. Patterson and John L. Hennessy. — *Computer Organization and Design, Revised Fourth Edition, Fourth Edition: The Hardware/Software Interface*. — 4th. — San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011. — ISBN: 0123747503.

12 H. Sutter. — “The Free Lunch Is Over A Fundamental Turn Toward Concurrency in Software”. — In: — 2013.

13 WOUTER SWIERSTRA. — “Data types à la carte”. — In: *Journal of Functional Programming* 18.4 (2008), p. 423–436. — DOI: 10.1017/S0956796808006758.

14 Philip Wadler. — “Theorems for Free!” — In: *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture*. — FPCA ’89. — Imperial College, London, United Kingdom: Association for Computing Machinery, 1989, — P. 347–359. — ISBN: 0897913280. — DOI: 10.1145/99370.99404. — URL: https://doi.org/10.1145/99370.99404.

15 Richard Zurawski and Mengchu Zhou. — “Petri net and industrial application: A tutorial”. — In: *Industrial Electronics, IEEE Transactions on* 41 (Jan. 1995), p. 567–583. — DOI: 10.1109/41.334574.

APPENDIX A. RUNFREEAP HOMOMORPHISM PROOF

Let us prove that runFreeAp e in Listing 15 is an applicative homomorphism.

Throughout the proof we'll use the the free theorem[14] for fmap on Listing A.1.

Listing A.1 – Free theorem for fmap specialized to FreeAp

```
runFreeAp e (fmap f m) = fmap f (runFreeAp e m)
```

By definition runFreeAp e is an applicative homomorphism if

1. runFreeAp e (pure x) = pure x
2. runFreeAp e (f' <*> g') = runFreeAp e f' <*> runFreeAp e g'

Statement 1 obviously follows from the first case of runFreeAp.

Statement 2 is proven by induction on f' size.

Induction basis: f' = PureA x = pure x.

pure x <*> g' = fmap x g', which follows from Functor-Applicative relationship, so LHS is runFreeAp e (f' <*> g') = runFreeAp e (fmap x g') = fmap x (runFreeAp e g'). The last equation follows from the free theorem.

runFreeAp e (pure x) = pure x follows from Statement 1, so RHS is runFreeAp e f' <*> runFreeAp e g' = fmap x (runFreeAp e g').

LHS = RHS, the basis is proven.

Now, let's prove the induction step. Suppose that the statement holds for all FreeAp with a lesser number of steps than f' and f' = ImpureA fx' n'.

Let us examine the LHS. In the following equation all equalities stem from equational reasoning. Note: f <$> x = fmap f x.

```
runFreeAp e (ImpureA fx' n' <*> g) =
= runFreeAp e (ImpureA fx' (fmap flip n' <*> g)) =
= fmap (\x f -> f x) (e fx') <*> runFreeAp e (fmap flip n' <*> g)
```

Now, fmap does not add to the size of n', so the proposition holds for it. Let fx = e fx', n = runFreeAp e n', g = runFreeAp e g' and let us rewrite the LHS further:

```
fmap (\x f -> f x) (e fx') <*> runFreeAp e (fmap flip n' <*> g') =
```

```
= fmap (\x f -> f x) fx <*> (fmap flip n <*> g)
```

Now, let us rewrite the RHS:

```
runFreeAp e f' <*> runFreeAp e g' =
= runFreeAp e (Impure fx' n') <*> g =
= fmap (\x f -> f x) fx <*> n <*> g
```

It is clear that the outermost function application of the LHS and the RHS are the same, it is `<*>`. So is the second argument — `g`. So it is enough to only show equality of the first arguments.
Below are lists of equivalent transformations of first arguments of the LHS and the RHS. Last expressions in lists are alpha-equivalent, so LHS = RHS and the proposition is proven.
The first statement in a list is a given, other statements follow from the previous statement for a reason listed at the bottom of the line in a comment. Reasons are either “law n” or “eq. r.”. “law n” means law #n from the list below, “eq. r.” stands for “equational reasoning or β-reduction”.

1. Functor composition law: `fmap (f . g) == fmap f . fmap g`
2. Applicative composition law: `pure (.) <*> u <*> v <*> w = u <*> (v <*> w)`
3. Functor-Applicative relationship: `fmap f x = pure f <*> x`
4. Applicative interchange law: `u <*> pure y = pure ($ y) <*> u`

The lists are given in the form of Haskell code to employ a type-checker as a safety measure. See Listing A.2

Listing A.2 – runFreeAp e is a homomorphism proof

```
lhsSeq, rhsSeq
  :: Applicative f => f t -> f (t -> a -> b) -> [f (a -> b)]

rhsSeq fx n =
  [ fmap (\x f -> f x) fx <*> n ]
  -- a given

lhsSeq fx n =
  [ (.) <$> fmap (\x f -> f x) fx <*> fmap flip n
  -- a given
  , (fmap (\f1 f2 -> f1 . f2) (fmap (\x f -> f x) fx))
    <*> fmap flip n
  -- eq. r.
  , fmap (\y -> (\f1 f2 -> f1 . f2) ((\x f -> f x) y)) fx
    <*> fmap flip n
  -- law 1
  , fmap (\y -> (\f1 f2 -> f1 . f2) (\f -> f y)) fx
    <*> fmap flip n
  -- eq. r.
  , fmap (\y f2 x -> f2 x y) fx <*> fmap flip n
  -- eq. r.
  , fmap (\y f2 x -> f2 x y) fx <*> (pure flip <*> n)
  -- law 3
  , (.) <$> fmap (\y f2 x -> f2 x y) fx <*> pure flip <*> n
  -- law 2
  , pure ($ flip) <*> ((.) <$> fmap (\y f2 x -> f2 x y) fx) <*> n
  -- law 4
  , fmap (\f -> f flip) (fmap (.) (fmap (\y f2 x -> f2 x y) fx))
     <*> n
  -- eq. r.
  , fmap ((\f -> f flip) . (.) . (\y f2 x -> f2 x y)) fx <*> n
  -- law 1
  , fmap (\x' ->
      (\f -> f flip) ((.) ((\y f2 x -> f2 x y) x'))) fx <*> n
  -- eq. r.
  , fmap (\x' ->
      (\f -> f flip) (\f1 -> (\f2 x -> f2 x x') . f1)) fx <*> n
  -- eq. r.
  , fmap (\x' -> (\f1 -> (\f2 x -> f2 x x') . f1) flip) fx <*> n
  -- eq. r.
  , fmap (\x' -> (\f2 x -> f2 x x') . flip) fx <*> n
  -- eq. r.
  , fmap (\x' f -> (\f2 x -> f2 x x') (flip f)) fx <*> n
  -- eq. r.
  , fmap (\x' f x -> flip f x x') fx <*> n
  -- eq. r.
  , fmap (\x' f -> f x') fx <*> n
  -- eq. r.
  ]
```