Concurrency with effects and handlers

Implementing concurrency with nondeterminism using algebraic effects and handlers

Arthur Jacques

Supervisor(s): Casper Bach Poulsen, Jaro Reinders

EEMCS, Delft University of Technology, The Netherlands

A Thesis Submitted to EEMCS Faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering
June 23, 2024

Name of the student: Arthur Jacques
Final project course: CSE3000 Research Project
Thesis committee: Casper Bach Poulsen, Jaro Reinders, Annibale Panichella

Abstract

Algebraic effects and handlers are a new paradigm in functional programming. They aim at modularly handling side effects, by separating the declaration of those effects, from how they are handled. In this paper, we show how we can leverage their use to create an interface for concurrency using algebraic effects for which we can prove a list of concurrency laws, and also design handlers that allow us to run programs concurrently, thereby demonstrating the practical application of algebraic effects and handlers in managing concurrency.

1 Introduction

A big challenge in the realm of computer science is the reconciliation of theoretical models with their practical implementations. While models provide a good framework for understanding and thinking about computational concepts, actual applied implementations of these models often introduce complexities and problems that diverge significantly from these models. This divergence can lead to implementations that seem too pragmatic or too distinct from their theoretical model, which makes the formal proof of their correctness challenging.

1.1 Algebraic effects and handlers

There is a relatively new paradigm in functional programming, called algebraic effects and handlers, which can help us with that issue. Algebraic effects and handlers are a modern, modular approach to managing side effects in programming, providing a structured and flexible framework for effectful computation. Unlike traditional methods that embed side effects directly within the core logic of functions, algebraic effects separate the description of effects from their implementation. This separation is achieved through defining effects abstractly in terms of their interface with the rest of the program, while handlers are used to concretely implement these abstract effects. It is this separation between effects and handlers that could help us reconcile theory with implementation.

1.2 Concurrency

In computer systems, one of the most challenging and essential areas where theory and practical implementation frequently diverge is in the management of concurrent processes, also known as concurrency. Concurrency involves the simultaneous execution of multiple tasks. This approach enables a computer system to perform various operations at once, either by interleaving multiple tasks on a single processor or by executing them in parallel across multiple processors. By using concurrency, systems can achieve better resource utilization, leading to increased efficiency and improved responsiveness in multitasking environments.

However, implementing concurrency is not without its challenges. The unpredictable nature of task execution order can result in race conditions, where the outcome of operations depends on the sequence and timing of uncontrollable events [11]. This can make software behavior difficult to predict and debug. Additionally, deadlocks are a common issue, occurring when multiple programs lock each other out by holding resources needed by the others. There is also the risk of data inconsistency, as simultaneous access to shared resources can lead to conflicts unless properly managed. These issues highlight the need for a properly designed interface for concurrency. In 1984, Bergstra and Klop published a paper, "Algebra of communicating processes" [3], in which they state a list of laws governing the nondeterministic behaviour of concurrent programs. This leads to the research question of this paper: **Can we write an interface for concurrency using algebraic effects and handlers that respects the concurrency laws from "Algebra of communicating processes" [3]?**

Using effects and handlers for implementing concurrency allows us to separate the interface from the implementation. We can use algebraic effects to clearly define the behaviour of concurrency and prove the concurrency laws, while handlers enable us to execute programs concurrently in various ways, whether deterministic or nondeterministic.

Concurrency has already been approached using effects and handlers. For example, in the paper "Asynchronous effects" [1], Ahman and Pretnar give an implementation of concurrency that uses signals and interrupts, where a signal from a sending program will be an interrupt for a receiving program. Another notable mention is the paper "A poor man's concurrency monad" [6], which does not involve algebraic effects but instead uses monads with continuations, and implements concurrency in a way similar to us. However, the implementation in this paper does not account for the nondeterminism of interleaving, which is one of the main focuses of our own.

In this paper, we make several technical contributions. We present in section 3 our interface for interleaving concurrency, which makes use of an effect for nondeterminism. To the best of our knowledge, this is the first of its kind. Then, in section 4, we prove that the laws from "Algebra of communicating processes" [3] hold for this interface. In section 5, we present a case study of the ABP model that makes use of our interface, which demonstrates the use of our interface in formalizing concurrent programs. The rest of the paper is structured as

follows. In section 2, we explain effects and handlers more in detail, as well as how we are covering concurrency. It also goes over the methodology of the project. In section 6, we explore the ethical implications of this project. In section 7, we discuss other papers that relate to the field or that try to achieve the same purpose as us. In section 8, we consider different concepts that are left unexplored by the project. Finally, in section 9, we summarize the work done and give direction for future improvements.

2 Background and Methodology

2.1 Algebraic Effects and Handlers

Before delving into the implementation, we look at the workings of algebraic effects and handlers we use. This implementation comes from Casper Bach Poulsen's blog post [15], which is itself based on the concepts introduced in the paper "Data types à la carte" [18]. The principle of an effect is based on the free monad, which we can define in Haskell as the datatype *Free*:

```
data Free f a = Pure a | Op (f (Free f a))
```

In which a *Pure* is a holder for a value of type *a*, while an *Op* holds an effect as *f*, which itself holds a *Free* recursively. Here is a possible example of an effect which could take the place of f:

```
data State s k = Put s k | Get (s -> k)
```

The *State* effect represents the access of the program to an external value. *s* represents the type of the value, while *k* represents the type of the continuation (which should be *Free*). Here is an example of a program implementing the *State* effect.

```
program = Op (Get (\s -> Op (Put 3 (Pure s))))
```

This code represents a program which uses that effect. To make this snippet look more like a program, let us make an instance of monad for the *Free* data type. The monadic bind uses a fold function which transforms the *Free* data type recursively when given a function to transform a *Pure* and a function to transform an *Op*.

```
instance Functor f => Monad (Free f) where
  m >>= k = fold k Op m
  return = Pure

fold :: Functor f => (a -> b) -> (f b -> b)
                  -> Free f a -> b
fold gen _   (Pure x) = gen x
fold gen alg (Op f) = alg (fmap (fold gen alg) f)
```

This allows us to write the previous program in the following way:

```
program = do s <- get; put 3;return s
```

Now we have a way to represent programs using algebraic effects. But how can we give implementations for handlers that will take care of these effects? We once again use the fold function, and give it a function that removes the effect handled, and changes the type held by *Pure* using the function that takes care of the *Pure*. Here is its implementation, an example of handler for *State*, as well as the result of handling our initial program using this handler with a given initial *state* of 0.

```
data Handler_ f a p f' b
  = Handler_ { ret_  :: a -> (p -> Free f' b)
             , hdlr_ :: f (p -> Free f' b)
                      -> (p -> Free f' b) }

handle_ :: (Functor f, Functor f')
        => Handler_ f a p f' b -> Free (f + f') a
        -> p -> Free f' b
handle_ h = fold
  (ret_ h)
  (\x -> case x of
     L x -> hdlr_ h x
     R x -> \p -> Op (fmap (\m -> m p) x))

hState :: Functor g
       => Handler_ (State s) a s g (a, s)
hState = Handler_
  { ret_ = \x s -> pure (x, s)
  , hdlr_ = \x s -> case x of
      Put s' k -> k s'
      Get k -> k s s }

handle_ hState program 0 == Pure (0, 3)
```

There are also simpler handlers that do not take an argument *p*.

This is the basic working of effects and handlers. More details are included in Poulsen's blog post [15].

2.2 Concurrency

As mentioned in the introduction, we use effects to make a model for concurrency that respects the laws from Bergstra and Klop's paper [3]. However, we base ourselves on the more recent version of the laws stated in the book "Modelling and Analysis of Communicating Processes" [7]. Figure 1 states all of those laws.

M	$x \parallel y = x \between y + y \between x + x \mid y$
LM1‡	$\alpha \between x = \alpha \cdot x$
LM2‡	$\delta \between x = \delta$
LM3‡	$\alpha \cdot x \between y = \alpha \cdot (x \parallel y)$
LM4	$(x + y) \between z = x \between z + y \between z$
LM5	$(\sum_{d:D} X(d)) \between y = \sum_{d:D} X(d) \between y$
S1	$x \mid y = y \mid x$
S2	$(x \mid y) \mid z = x \mid (y \mid z)$
S3	$x \mid \tau = x$
S4	$\alpha \mid \delta = \delta$
S5	$(\alpha \cdot x) \mid \beta = \alpha \mid \beta \cdot x$
S6	$(\alpha \cdot x) \mid (\beta \cdot y) = \alpha \mid \beta \cdot (x \parallel y)$
S7	$(x + y) \mid z = x \mid z + y \mid z$
S8	$(\sum_{d:D} X(d)) \mid y = \sum_{d:D} X(d) \mid y$
TC1	$(x \between y) \between z = x \between (y \parallel z)$
TC2	$x \between \delta = x \cdot \delta$
TC3	$(x \mid y) \between z = x \mid (y \between z)$

Figure 1: Concurrency laws [7]

The figure makes use of a lot of different operators, of which it is important we know the meaning:

- The symbol for alternative composition, $+$, represents the nondeterministic choice between two alternatives.
- $\parallel$ is the symbol for concurrency.
- The leftmerge operator, $\between$, represents the concurrency of two programs, with the program on the left having the priority (the first action to take place will be an action from the left program)
- The α symbol represents a program composed of a single operation.
- The δ symbol represents a deadlock.
- The $\cdot$ operator indicates a sequential composition.

We look more closely at each of those laws in section 4. This model focuses on the nondeterminism of concurrency, which has to do with the randomness of the order of operations from two concurrent programs. Figure 2 shows this nondeterminism illustratively.

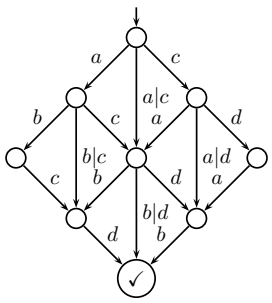

Figure 2: Nondeterminism in concurrency [7]

Our model follows the concurrency laws and implements interleaving concurrency, so it is not parallel in any way. We discuss what we mean by interleaving concurrency more in detail in section 8.

2.3 Methodology

In order to implement the interface, we expanded upon the code snippets given in Poulsen's blog post [15], which is all written in Haskell, with which we were able to create relevant effects and handlers. We used equational reasoning for proving the laws, and wrote the proofs in text files. Both the implementation and the proofs are available in the repository of the project.

3 Implementation

How can we simulate concurrency by interleaving two programs represented by free monads? Since the free monad uses continuation-passing style, the answer to this question is very straightforward, and is illustrated in the following snippet.

```
a = Op f (Op g (Pure x))
b = Op h (Op i (Pure y))
par a b = Op (f (Op h (Op g (Op i (Pure (x,y))))))
```

The rest of this chapter goes over the implementation of the model for concurrency. First, we define the behaviour of our interface using the laws. We then implement the effect to represent nondeterminism. Finally, we give our interface for concurrency.

3.1 Defining our interface through the laws

In order to define our interface, we first state what the laws entail. We first focus on the laws M through LM3. The first law M describes the behaviour of the concurrency of two programs x and y as the conjunction of the case where x is prioritized, the case where y is prioritized, and the case where both x and y go simultaneously (which we ignore for our interface). The law LM1 and LM3 describes better the behaviour of the leftmerge. LM1 states that if a single-action program has priority over another program, it is equivalent to the action of the first program followed by the second program, while LM3 states that a program that has priority over another program with which it is run concurrently is equivalent to the first operation of that program followed by the rest of the program ran concurrently with the other program. This law is very important as it shows the recursive step involved in concurrency. The law LM2 describes the behaviour of a deadlock. It states that if a deadlock has priority

over a program, the resulting concurrent program is a deadlock as well, as the program is then halted.

The important features to consider for our interface are thus the plus operator, the leftmerge operation, and the deadlock.

3.2 Implementing the plus operator

As mentioned in the introduction, concurrency comes with some inherent nondeterminism, coming from the interleaving of the actions of the program. In the concurrency laws, that nondeterminism is handled by the plus operator, which describes the division into different cases. This can be implemented using the *Choose* effect taken from the paper "Handlers in action" [8]. However, in the paper, *Choose* is given both continuations, which we cannot do using the version of free monad that we use. We hence use the same implementation as the *amb* effect from Leijen's paper [10], and then use that effect to build the *plus* operator which we can later use. Leijen wrote it in Koka, so we need to adapt it in Haskell. The effect *Failure* from "Handlers in action" is transformed into *Zero*, which corresponds to the deadlock from the laws.

```
data Choose k
  = Choose (Bool -> k) | Zero
  deriving Functor

choose :: Choose <: f => Free f Bool
choose = Op (inj' (Choose Pure))

zero :: Choose <: f => Free f a
zero = Op (inj' Zero)

(~+~) :: Choose <: f => Free f a -> Free f a
                     -> Free f a
m1 ~+~ m2 = do
  b <- choose
  if b then m1 else m2
```

We then implement the equivalent of the *AllResults* handlers from "Handlers in action" [8], which we call *hChoose'*, and which accumulates the results of all the nondeterministic branches and discards any failing continuation.

```
hChoose' :: Functor f' => Handler Choose a f' [a]
hChoose' = Handler
  { ret = \x -> pure [x]
  , hdlr = \case
      Choose f -> f False >>= \l
               -> fmap (++ l) (f True)
      Zero -> pure []}
```

3.3 Interface for concurrency

With the plus operator implemented, we now give our interface for concurrency:

```
par :: Choose <: f => Free f a -> Free f b
                   -> Free f (a, b)
par (Pure x) y = fmap (x,) y
par x (Pure y) = fmap (,y) x
par x y = do
  goesFirst x y ~+~ fmap swap (goesFirst y x)

goesFirst :: Choose <: f => Free f a -> Free f b
                         -> Free f (a, b)
goesFirst (Pure x) y = par (Pure x) y
goesFirst (Op x) y = Op (fmap (`par` y) x)
```

The function *par* works by using the *Choose* effect to non-deterministically decide which of the two programs to prioritize. This prioritization, which is the same as the leftmerge operation from the concurrency laws, is encapsulated in the function *goesFirst*, which calls *par* recursively with the continuation of the prioritized program and the entirety of the other one.

We then look at some use of this interface. To do so, we need to use effects that make sense to use with concurrency. Let's use the *State* effect, as introduced in the paper "Handling Algebraic Effects" [14]. This one makes sense to use in the context of concurrency, as it represents a shared state between the two threads. We also use the Exception effect, developed in the same paper [14]. Implementations of both *State* and *Err* (Exception) as well as their handlers are however taken from Poulsen's blog post [15]. Consider the following programs:

```
program1 :: (Err <: f, State Int <: f)
         => Free f Int
program1 = do
  (s::Int) <- get'
  put' (s + 4)
  (s::Int) <- get'
  if s > 7 then err' "foo" else Pure s

program2 :: (Err <: f, State Int <: f)
         => Free f Bool
program2 = do
  (s::Int) <- get'
  put' (s+5)
  (s::Int) <- get'
  Pure (s > 7)

pairing :: (Err <: f, Choose <: f, State Int <: f)
        => Free f (Int, Bool)
pairing = par program1 program2

un(handle hChoose' (handle_ hState'
    (handle hErr triplepairing) [0::Int]))
```

We use a function that counts the amount of time that each outcome happens, which results in the following list of all possible concurrent orderings of the two programs:

```
(Right (4,True),[9,4,0]): 2
(Left "foo",[9,4,0]): 3
(Right (4,False),[4,5,0]): 6
(Right (4,False),[5,4,0]): 2
(Right (5,False),[5,4,0]): 4
(Left "foo",[9,5,0]): 5
"length"
22
```

4 Proving the laws

4.1 The laws

In section 3.1, we have explained a few of the laws stated in "Modelling and Analysis of Communicating Systems" [7]. In this chapter, we aim at proving that those laws and the others hold for our interface. However, we cannot prove them all. Law LM5 is a generalization of LM4 but with a list of unknown size of program ran concurrently, which is not how our interface works. The laws S1-8 and TC3 are concerned with operations run simultaneously, which our interface does not cover. The rest of the laws can be proven to hold for our interface.

The laws can be rewritten using the interface in the following way:

```
M: par m1 m2
 == goesFirst m1 m2 ~+~ fmap swap (goesFirst m2 m1)
LM1: goesFirst (Op (f (Pure x))) m
 == Op (f (Pure x)) >>= (\x -> fmap (x,) m)
LM2: goesFirst zero m
 == zero
LM3: goesFirst (Op (f (Pure a)) >>= x) y
 == Op (f (par (Pure a >>= x) y))
LM4: goesFirst (x ~+~ y) z
 == goesFirst x z ~+~ goesFirst y z
TC1: goesFirst (goesFirst x y) z
 == goesFirst x (par y z)
TC2: goesFirst x (Op Zero) == x >>= (Op Zero)
```

Apart from those, it would also make sense to prove commutativity and associativity for the *par* function, which looks as follows:

```
Commutativity: par x y == fmap swap (par y x)
Associativity: par x (par y z)
 == fmap assoc (par (par x y) z)
```

4.2 Proofs

The law M holds trivially, as it is simply a one step unfolding of the function *par*. The proof for LM1 goes as follow:

```
  goesFirst (Op (f (Pure x))) m
= {by definition of goesFirst}
  Op (f (par (Pure x) m))
= {by definition of par}
  Op f (fmap (x,) m)
```

Here is the proof for LM2:

```
  goesFirst zero m
= {by definition of zero}
  goesFirst (Op Zero) m
= {by definition of goesFirst}
  Op (fmap (`par` m) Zero)
= {by definition of fmap for Zero}
  Op Zero
= {by definition of zero}
  zero
```

The proof for LM3 holds trivially. The proof relied on a change in the function *goesFirst* which makes it recursively call itself instead of *par* when encountering a *Choose* effect. We do not show that change in code here, but it is available in the repository.

```
  goesFirst (x ~+~ y)
= {by unfolding the plus operator}
  goesFirst (Op (Choose (\k -> if k then x
                                  else y))) z
= {by applying fmap on the function
   of the choose with the goesFirst}
  Op (Choose (\k -> if k then goesFirst x z
                         else goesFirst y z))
= {by folding the plus operator back}
= goesFirst x z ~+~ goesFirst y z
```

The proof for TC1 reduces to the proof for associativity of *par*, which we cover later.

```
LHS
  fmap assoc (goesFirst (goesFirst (Op (f k)) y) z)
= {by unfolding the inner goesFirst}
  fmap assoc (goesFirst (Op (f (par k y))) z)
= {by unfolding goesFirst}
  fmap assoc (Op (f ( par (par k y) z)))
= {by using lemma b}
  Op f (fmap assoc (par (par k y) z))
RHS
  goesFirst (Op (f k)) (par y z)
= {by unfolding the goesFirst}
  Op (f (par k (par y z)))
```

For the proof for TC2, we used induction, as well as the instance of alternative for the Free monad, as expressed in chapter 5 of the paper "Freer Monads, More Extensible Effects" [9], which we adapt to our implementation.

```
instance (Functor f, Choose <: f)
    => Alternative (Free f) where
  empty = zero
  (<|>) = (~+~)

Induction step:
LHS
```

```
  goesFirst (Op (f k)) (Op Zero)
= {by unfolding goesFirst}
  Op (f (par k (Op Zero)))
= {by unfolding the par}
  Op (f (goesFirst k (Op Zero)
    ~+~ fmap swap (goesFirst (Op Zero) k)))
= {by unfolding the second goesFirst}
  Op (f (goesFirst k (Op Zero)
    ~+~ fmap swap (Op Zero)))
= {by using the alternative instance}
  Op (f (goesFirst k (Op Zero)))
RHS
  (Op (f k) >>= zero)
= {using lemma a}
  Op (f (k >>= zero))

Base case:
LHS
  goesFirst (Pure x) (Op Zero)
= {unfolding the goesFirst}
  fmap (x,) (Op Zero)
= {applying the fmap}
  Op Zero
RHS
  (Pure x) >>= zero
= {by definition of >>=}
  Op Zero
```

4.3 Properties of the plus operator

In order to prove the associativity and the commutativity of the *par* function, we first need to prove them for the plus operator. To do so, we use the handler *hChoose'*, defined in section 3. If the lists of accumulated results from applying *hChoose'* to two free monads contain the same entries, regardless of the order or frequency, the two free monads are considered equal. Unfortunately, this means that we can only prove properties of the plus operator if this handler is applied to a program with only the *Choose* effect, as we cannot predict the repercussions of other arbitrary effects. This translates to having to apply the handler for *Choose* last for these properties to hold. Because of the way handlers work, this means that we will only have to prove those properties for free monads only composed of the *Choose* effect, and of a *Pure*. We do not show the proofs here, but one can find them in the repository of the project.

4.4 Commutativity and Associativity

Using the conditional commutativity and associativity of the plus operator, we can prove the commutativity and associativity of the *par* function. Commutativity turns out to be pretty straightforward and can be proven as follows.

```
LHS
  par m1 m2
= {unfolding the par}
  goesFirst m1 m2 ~+~ fmap swap (goesFirst m2 m1)
RHS
  fmap swap (par m2 m1)
= {unfolding the par}
  fmap swap (goesFirst m2 m1
    ~+~ fmap swap (goesFirst m1 m2))
= {using lemma p}
  (fmap swap (goesFirst m2 m1)
    ~+~ (fmap (swap . swap) (goesFirst m1 m2)))
= {by cancelling the swap's}
  (fmap swap (goesFirst m2 m1)
    ~+~ (goesFirst m1 m2))
= {by using the commutativity of the ~+~}
  goesFirst m1 m2 ~+~ fmap swap (goesFirst m2 m1)
```

The proof for associativity was more intricate. We used structural indexing, to prove that the inductive step, where all three programs are of the form *Op (f k)*, calls recursively three other cases, for each of which one of the programs turns into a Pure (the cases are separated by the + operator, and we make use of its commutativity and associativity). Those three cases each recursively call two other cases, for each of which one of the remaining *Op's* turns into a *Pure*. It also uses LM4 as a lemma. We do not show the actual proof in this paper. It is however available in the repository.

5 Applications

We have now proven that our interface respects the laws from the book. We can now ask: What is a possible use for this interface? In "Modelling and analysis of communicating systems" [7], the author gives an example of a real-world program that implements concurrency, the ABP model:

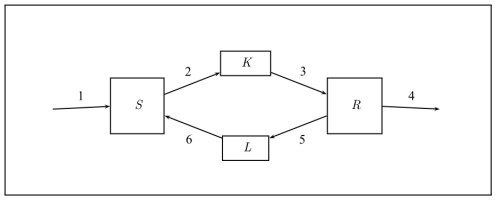

Figure 3: Illustration of the ABP Model [7]

The ABP model ensures reliable data transmission from sender (S) to receiver (R) over lossy channels. The sender transmits data packets with an alternating control bit via channel K. The receiver processes the packet if the control bit matches the expected sequence and sends an acknowledgment (ACK) through channel L. Upon receiving the correct ACK, the sender switches the control bit and sends the next packet. If the ACK is not received,

the sender retransmits the packet, ensuring data integrity and reliability despite packet loss or duplication. Bergstra's paper demonstrates that this approach is valid [4]. We now provide an implementation of this algorithm using our interface. To achieve this, we first need to address an effect related to concurrency, which is locking.

5.1 Locking

To implement locking, we make use of a new effect, *Lock*:

```
data Lock k = Lock  k | Unlock k
```

Then we need a new modified version of the *par* function that will handle those effects differently from others, in that it "locks" on the current prioritized process in case of a *Lock*, and goes back to normal when encountering an *Unlock*. The handler for *Lock* simply removes it from the program, and is only applied to the program when the *par* function has already been applied, or when you do not want the locking mechanism to take action any longer.

This feature allows us to group effects together and guarantee them to follow each other, even if the program they are a part of is run concurrently with some other program.

5.2 ABP Model

Now that we have a locking mechanism in place, we can give an implementation of the ABP model. We do not show the code, but instead explain how we went about it. It consists of two programs run concurrently using *par*, one of which represents the sender, and the other the receiver. Each of those is itself a function taking as a parameter a boolean representing the internal state of the process. Both lossy channels are represented using a *State*, one containing a boolean and a chunk of data for channel K and one containing simply a boolean for channel L. So the communication as explained in the book is covered by a simple *put* or *get*. The problem of not having communication through simultaneity of actions is further explored in section 8. The lossiness of the channels is covered by the use of the *Choose* effect through the function *par*, where a program being prioritized translates to a successful transmission to or from the channel, and then the error message is not sent, which we consider equivalent to the state not changing. In the case where one of the booleans contained in the *States* representing the respective channels tells the program that no action is required (silent step), the same program is called recursively. A proof of the validity of this implementation is available on the repository of the project.

This implementation of the ABP Model demonstrates the use of our interface in modelling concurrent programs. It allows us to prove properties using equational reasoning, while also providing the possibility of using handlers to execute these programs. This approach showcases the versatility of algebraic effects and handlers in both theoretical and practical contexts.

6 Responsible research

The code written for this implementation as well as the proofs for the laws have been made public through the use of GitHub, at the address https://github.com/Arthur158/concurrency_effects. The methodology for writing this code has also been described, and the sources containing the code on which we base ourselves have also been stated.

This paper has as its goal to explore the use of algebraic effects and handlers in formalizing the integrity of concurrency models, as well as giving an implementation to be able to use the given interface. The highly theoretical nature of the subject and the current lack of practical applications of our interface makes the given implementation seemingly harmless, and we could not think of a way it could be used with malicious intent.

7 Related work

The theoretical foundations of algebraic effects and handlers were first given by Plotkin and Power, who introduced algebraic operations to model side effects within programming languages [12]. This work was further developed by Plotkin and Pretnar, who explored the management of these effects through handlers, advancing the practical application of the theory [13]. The practical implications and implementations of these theories have been demonstrated by Bauer and Pretnar, whose work on the Eff programming language showcases the utilization of algebraic effects and handlers to enhance software modularity and reusability [2]. These foundational studies have established a robust framework that continues to influence contemporary research in handling computational effects in functional programming.

In the introduction, we mentioned the paper "Asynchronous effects" [1], which also aims at handling concurrency using algebraic effects and handlers. However, it does so in a very different fashion. While our interface works by interleaving the steps to simulate the random order of execution, their implementation instead keeps the programs separate, and lets those programs send signals to one another. Those signals will then interrupt the

other program for it to react in some fashion. This approach provides a much more practical way to handle concurrency, but it does not relate to the concurrency laws, as it does not have the notion of interleaving.

The paper "A poor man's concurrency monad" [6] gives an implementation much closer to our own. In it, Claessen also decided to simulate concurrent processes by interleaving them. When talking about how to suspend a process, he also mentions the need to "grab the future and stick it away for later use", and then mentions continuation passing style as a way to do so. This resembles the behaviour of algebraic effects, and the way our interface works. It however differs in that it does not account for the nondeterminism inherent to concurrency. It instead iterates over all processes run concurrently and adds new forked processes or continuations at the end of the list. Our interface makes use of the Choose effect to represent that nondeterminism, and delegates its execution to handlers.

8 Discussion

We now have an interface for concurrency, proven to respect some concurrency laws, as well as a working implementation of the ABP model that is also proven to work correctly. in this section, we will discuss different aspects in which this implementation lacks, as well as how it relates to the literature.

8.1 Concurrency as an effect

Our interface for concurrency is based on the function *par*, which combines two programs represented by free monads into a single program by using the *Choose* effect to delegate the handling of nondeterminism. However, an alternative to this approach would be to model concurrency itself as an effect, for which we could then design a handler to execute concurrency whichever way we want. However, as is explored in the paper "Hefty Algebras: Modular Elaboration of Higher-Order Algebraic Effects" [16], we would run into the problem that that effect would be a higher-order effect, which the implementation of the free monad that we used does not support. In the paper, Poulsen and Van der Rest also develop a solution to this problem in the form of *Hefty Algebras*.

8.2 Interleaving concurrency

The main problem with this interface, as mentioned in the section 2, is that it implements interleaving concurrency, and not true concurrency. Interleaving concurrency is a model of concurrent execution where multiple tasks or processes are executed by alternating between them, giving the appearance of simultaneous execution. Our interface, as implemented, does not effectively reduce the execution time when running programs concurrently. This mostly stems from how the laws we used treated concurrency. They treat each program as a list of atomic operations, and two programs run concurrently as an interleaving of those. However, true concurrency involves a different approach in which atomicity is not assumed, as mentioned in the paper "Concurrency vs Interleaving, an instructive example" [5].

8.3 Lack of simultaneity

A part of the concurrency laws that we chose to ignore is the part concerning simultaneity. In the book [7], simultaneity of two execution steps from different programs can represent communication between those two processes.

Now let us consider effect fusion. Fusion between effects has already been explored in the literature, for example in the paper "Reasoning about effect interaction by fusion" [19] as well as more briefly in the paper "Effect handlers in scope" [17]. It entails handlers of several different effects "merging", to create a theory that encompasses all the subtheories. Since our concurrency interface works by interleaving different effects, such a fusion of effects could be an interesting way to tackle the lacking simultaneity.

9 Conclusion and future work

In this paper, we considered the laws relating to concurrency as given in the paper "Algebra of communicating processes" [3], and used those to write an interface for concurrency using algebraic effects and handlers, using the *Choose* effect to represent the nondeterminism involved. Once that interface was functional, we proved that those laws held for our interface using equational reasoning. We then used this interface to give a working implementation of the ABP model, and were then able to prove its integrity.

There are different ways in which the work done here could be pursued. Currently, the interface is based on a function, but hefty algebras can be explored to make concurrency into an effect. One could also try to find a way to make it more practical, by basing it on true concurrency. However, we are here limited by the nature of the laws we used, which assume atomicity of actions. Another possible direction would be to explore the resemblance between simultaneity as defined in the concurrency laws and the concept of fusion of effects. This could make the interface more thorough, as we could then potentially cover all the concurrency laws.

10 Acknowledgements

We would like to thank our professor and supervisor for the support given throughout this project. Their expertise on the subject was really useful to help us progress with our research. Their insightful feedback and constructive criticism contributed greatly to the depth and quality of our work.

References

[1] Danel Ahman and Matija Pretnar. Asynchronous effects. Research report, University of Ljubljana, Ljubljana, Slovenia, 2020.

[2] Andrej Bauer and Matija Pretnar. Programming with algebraic effects and handlers. *Journal of Logical and Algebraic Methods in Programming*, 84(1):108–123, 2015.

[3] J. A. Bergstra and J. W. Klop. Algebra of communicating processes. Technical report, Centre for Mathematics and Computer Science, P.O. Box 4079, 1009 AB Amsterdam, The Netherlands, 1984.

[4] J.A. Bergstra and J.W. Klop. Verification of an alternating bit protocol by means of process algebra. *Centre for Mathematics and Computer Science*, 1986.

[5] Luca Castellano, Giorgio De Michelis, and Lucia Pomello. Concurrency vs interleaving: An instructive example. *Bulletin of the European Association for Theoretical Computer Science (EATCS)*, 31, 01 1987.

[6] KOEN CLAESSEN. A poor man's concurrency monad. *Journal of Functional Programming*, 9(3):313–323, 1999.

[7] Jan Friso Groote and Mohammad Reza Mousavi. *Modelling and Analysis of Communicating Systems*. MIT Press, Cambridge, MA, USA, 2014.

[8] Ohad Kammar, Sam Lindley, and Nicolas Oury. Handlers in action. *Unknown Journal*, Unknown Year. University of Cambridge; University of Strathclyde.

[9] Oleg Kiselyov and Hiromi Ishii. Freer monads, more extensible effects. *SIGPLAN Not.*, 50(12):94–105, aug 2015.

[10] Daan Leijen. Type directed compilation of row-typed algebraic effects. *SIGPLAN Not.*, 52(1):486–499, 2017.

[11] Catuscia Palamidessi. Nondeterminism and concurrency. `http://www.lix.polytechnique.fr/Labo/Catuscia.Palamidessi/Talks/100315-ENS.pdf`, 2010. Talk presented at ENS.

[12] Gordon D Plotkin and John Power. Algebraic operations and generic effects. *Applied Categorical Structures*, 11(1):69–94, 2003.

[13] Gordon D Plotkin and Matija Pretnar. Handlers of algebraic effects. In *European Symposium on Programming*, pages 80–94. Springer, 2009.

[14] Gordon D Plotkin and Matija Pretnar. Handling algebraic effects. *Logical Methods in Computer Science*, Volume 9, Issue 4, December 2013.

[15] Casper B. Poulsen. Algebraic effects in practice: Theory and implementation, 2023. Accessed: 2024-05-14.

[16] Casper Bach Poulsen and Cas van der Rest. Hefty algebras: Modular elaboration of higher-order algebraic effects. *Proceedings of the ACM on Programming Languages*, 7(POPL):1801–1831, 2023.

[17] Tom Schrijvers, Exequiel Rivas, Alexander Vandenbroucke, and Maarten Bruynooghe. Effect handlers in scope. *Journal of Functional Programming*, 25(4-5):856–881, 2015.

[18] WOUTER SWIERSTRA. Data types à la carte. *Journal of Functional Programming*, 18(4):423–436, 2008.

[19] Zhixuan Yang and Nicolas Wu. Reasoning about effect interaction by fusion. *Proc. ACM Program. Lang.*, 5(ICFP), aug 2021.