State will do

Willem Seynaeve *[1], Koen Pauwels **[1], and Tom Schrijvers[1]
Category: Research Article (extended abstract)

Department of computer science KULeuven Belgium
willem.seynaeve@student.kuleuven.be

Abstract. The main strength of pure languages like Haskell is that they allow a straightforward way of reasoning about programs called equational reasoning. Gibbons and Hinze propose an axiomatic approach for monadic equational reasoning which uses laws to characterise effects. In this paper they show how the state monad and the nondeterminism monad can be combined in one effect called backtrackable state (or "local state") and they illustrate how the two effects interact. A second way for state and nondeterminism to interact is non-backtrackable state (or "global state") where state persists after backtracking. Pauwels et al. show how backtrackable state can be simulated with non-backtrackable state and prove correctness using the axiomatic approach proposed by Gibbons and Hinze. Within this project we take this approach one step further and simulate the backtrackable state (state and nondeterminism) with only the state effect.

Keywords: monads · equational reasoning · nondeterminism · state

1 Introduction and background

One of the appeals of purely functional programs is that they possess the property that equals may always be substituted for equals [5], which makes reasoning about these programs more straightforward. This power comes at a disadvantage: our functions must be free of side effects, which makes some programs (for example, stateful or nondeterministic programs) harder to express. Wadler [6] shows how such effects can be encapsulated using the `Monad` interface. In Haskell the monad class looks as follows:

```
class Monad m where
  return :: a   -> m a
  (>>=)  :: m a -> (a -> m b) -> m b
```

This class is accompanied by three laws, which are rules we want any implementation of the monad interface to obey.

* Student
** Phd Student

```
--Left identity:
return a >>= f  = f a
--Right identity:
m >>= return    = m
--Associativity:
(m >>= f) >>= g = m >>= (\x -> f x >>= g)
```

So, with monads as an interface to implement side effects, we can now have much cleaner code while our functions are still pure, so we can use equational reasoning on our programs. One method of reasoning about effectful programs is to expand then out and to prove properties about them. This again is a tedious task. Hutton and Fulger [2] show how one can reason about effectful programs without the need for this expansion. Gibbons and Hinze [1] go even further. Instead of reasoning about a concrete implementation of an effect, they present an axiomatic approach: by exploiting algebraic properties of a monadic interface, they show that one can write equational proofs which preserve the monadic abstraction (in other words, the proof is entirely independent on the concrete implementation of an effect). They also show that this axiomatic approach works for combinations of effects, which they illustrate on a "backtrackable state" effect (also called "local state"), defined as a combination of nondeterminism and state with laws that stipulate the backtracking behaviour. Pauwels et al. [4] build upon the work of Gibbons and Hinze by documenting the laws that give rise to *non-backtrackable* state (also called "global state"), and using the same axiomatic reasoning technique to show that the backtrackable state interface can be implemented in terms of the non-backtrackable state interface. In this work, we take this approach one step further and simulate the backtrackable state effect with only state.

2 Simulating Nondeterminism and State with just State

In this project, we aim to simulate a "stateful nondeterminism" effect (more specifically *backtrackable state*) with a "lower level" implementation that only uses state. Furthermore, we wish to prove this simulation correct. Eventually we will use the approach proposed by Gibbons and Hinze to prove one interface can be simulated by the other, but we first explore a Hutton and Fulger style proof, based on concrete implementations for both effects. The concrete proof can then be a guide for finding the minimal set of laws we have to impose on the relevant interfaces to prove the simulation. Eventually this project should lead to a proof of the simulation in Coq[1].

2.1 Motivational Example

One could ask why is a simulation of the stateful nondeterminism monad with the state monad useful. The backtrackable state effect [1] is a setting that feels natural for implementing naive straightforward backtracking algorithms. Say,

[1] https://coq.inria.fr

for example, that you want to write a Prolog interpreter, and prove it correct. If you were to write this program only in terms of "low level" effects such as state, it would be a tedious task to prove the correctness of your algorithm, because you have to take into account a lot of the low level "plumbing". Having read Hutton and Fulger [2], one knows that a higher level effect shields away that complexity allowing you to reason about the program without having to take into account the low level plumbing. So you write an implementation with a backtrackable state effect which will be relatively easy to prove correct. The problem with this implementation would be that it would most likely not be very efficient. To optimize our conceptual interpreter we would like to have a more low level implementation, like one using only the state monad. So we want a backtrackable state implementation to reason about the program and a state implementation to have the tools to optimize our implementation. If we can simulate the backtrackable state monad with the state monad, we allow the programmer to implement the high level interpreter, simulate it with state, so that he have a low level implementation, and from there he can optimize. If every optimization is correct, the optimized interpreter is correct.

2.2 Simulating Nondeterminism with State

To illustrate how such a simulation might work, this section provides a simulation of an implementation of nondeterminism using an implementation of state. The `Monad` class for our implementation of nondeterminism `Nondet` is:

```
data Nondet a = Ret a
              | Fail
              | Nondet a :| Nondet a

instance Monad [] where
  return x = Ret x
  m >>=  f = case m of
    Ret a  -> Ret f a
    Fail   -> Fail
    a :| b -> (f >>= a) :| (f >>= b)
```

Note that for the implementation we used free Monads [3]. To transform our `Nondet` program into a stateful program, we need the programs to be data. For this reason, we make the `Nondet` effect a data type and have an explicit `runNondet` function.

```
runNondet :: Nondet a -> []
runNondet (Ret x)   = [x]
runNondet Fail      = []
runNondet (a :| b) = (runNondet a) ++ (runNondet b)
```

For State, we also have a free monad implementation, accompanied by a `runState` function.

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

runState :: State s a -> (s -> (s, a))
runState (Get f)    = \s -> runState (f s) s
runState (Put st x) = \s -> runState x st
runState (Return x) = \s -> (s, x)

instance Monad (State s) where
  return x = Return x
  st >>= f = case st of
    Get k    -> Get (\s -> ((k s) >>= f) )
    Put s m  -> Put s (m >>= f)
    Return x -> f x
```

2.3 The Simulation

To simulate the `Nondet` effect, we need a `state` with two components: a list where we store the results calculated during the computation and a stack storing what we still have to do. This stack is thus a stack of stateful subprograms. This recursive type of stateful programs containing stateful programs is encapsulated in the type `Prog a`. We also provide a few helper functions to modify the State variables namely, push and pop, that modify the Stack, and emit that adds a solution to the list of solutions. This brings us to the following transformation:

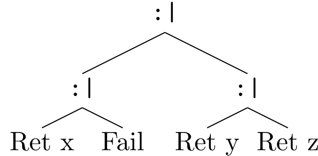
```
newtype Prog a = Prog (State ([a],[Prog a]) ())

emit :: a -> Prog a -> Prog a
emit x (Prog k) = Prog $ Get $ \(xs,stack) -> Put (xs++[x],stack) k

push :: Prog a -> Prog a -> Prog a
push p (Prog k) = Prog $ Get $ \(xs,stack) -> Put (xs,p:stack) k

pop :: Prog a
pop = Prog $ Get $ \(xs,stack) -> case stack of
  []            -> Return ()
  ((Prog p):ps) -> Put (xs,ps) p

trans :: Nondet a -> Prog a
trans Fail       = pop
trans (Ret x)    = emit x pop
trans (p :| q)   = push (trans q) (trans p)
```

The function `emit` adds a solution, the function `push` pushes a stateful program onto the stack and the function `pop` pops one element of the stack, unless the

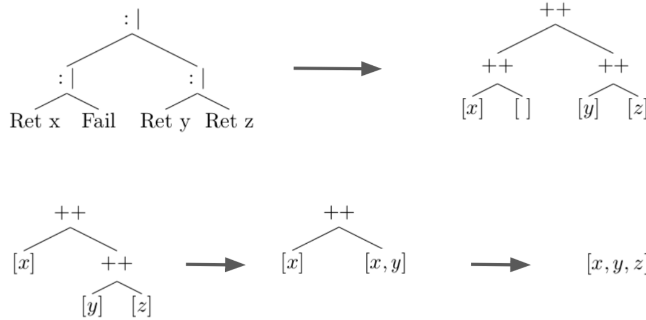
stack is empty, then the program returns. For example, consider the following Nondet program:

```
(Ret x :| Fail) :| (Ret y :| Ret z)
```

Or in tree form:



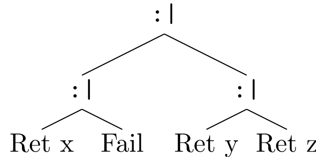
Evaluating with runNondet Running this program with the `runNondet` function recursively evaluates the two branches of the root `(:|)`, and concatenates the results. A visual representation of the evaluation is given below.



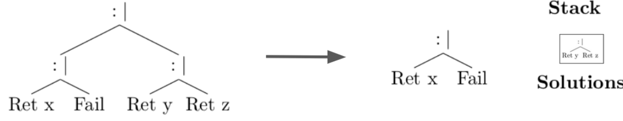
Evaluating with State To simulate the Nondet effect in a program with two stateful variables. A stack that serves as a continuation and a list of solutions found so far. We define the "runNondet" function as:

```
runNondet' nd (sol, stack) = fst $ fst $ runProg (trans nd) $ ([], [])

runProg :: Prog a -> (([a],[Prog a]) -> (([a],[Prog a]), ()))
runProg (Prog x) = runState x
```

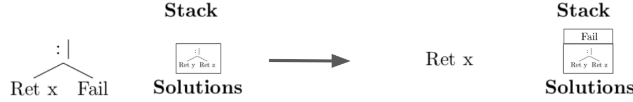
The simulation happens in the trans function. In our example, we first observe the program



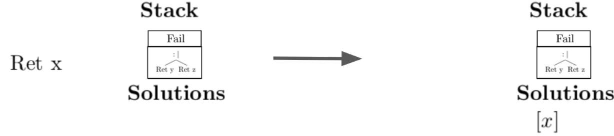
This is simulated by first translating the two branches, then handling the left-hand side and pushing the right-hand side onto the stack to be executed later.



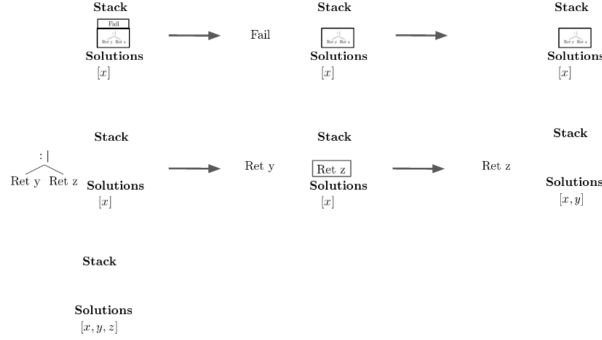
When we execute the left-hand side the same pattern has to be handled so we again push the right branch on the stack and execute the left-hand branch.



Now we reach the leaf `Ret x`, this adds one element to the list of solutions.



Now, there is nothing more to be done, so we pop from the stack to execute what we did not yet:



We see the result we want from the `Nondet` effect as the eventual list of solutions, namely $[x, y, z]$.

2.4 Proof of correctness

Theorem: Say `x : Nondet n`, then:

```
runNondet' x ([] , [])
=
runNondet x
```

Proof: The two cases `x = Fail` and `x = Ret x` are handled by lemma's 2 and 3 respectively. So all that is to prove is: **x = a :| b**. This is just a consequence

of lemma 4, with [] and [] as the initial solutions and stack.
Q.E.D

Lemma 2:

```
runNondet' Fail ([], [])
=
[]
```

Proof:

```
runNondet' Fail ([], [])
= --definition of runNondet'
fst $ fst $ runProg (trans Fail) $ ([], [])
= --definition of trans
fst $ fst $ runProg (Prog pop) $ ([], [])
= --definition of runProg
fst $ fst $ runState pop $ ([], [])
= --definition of pop
fst $ fst $ runState (Return ()) ([], [])
= --definition of runState
[]
```

Lemma 3:

```
runNondet' (Ret x) ([], [])
=
[x]
```

Proof:

```
runNondet' (Ret x) ([], [])
= --definition of runNondet'
fst $ fst $ runProg (trans (Ret x)) $ ([], [])
= --definition of trans
fst $ fst $ runProg (Prog (emit x pop)) $ ([], [])
= --definition of runProg
fst $ fst $ runState (emit x pop) $ ([], [])
= --definition of emit
fst $ fst $ runState pop ([x], [])
= --definition of pop
fst $ fst $ runState (Return ()) ([x], [])
= lemma
[x]
```

Lemma 4: Say x,y is of type (Nondet n), sol is of type [n] and stack is of type (Prog a), then:

```
runNondet' (x :| y) (sol , stack)
=
sol ++ runNondet' x ([], []) ++ runNondet' y ([], stack)
```

Proof:

- **case:** x = Fail

```
runNondet' (x :| y) (sol, stack)
= --lemma 5
runNondet' Fail (sol, ((Prog $ trans y):stack)
= --def runNondet' and trans
fst $ fst $ runProg pop (sol, (Prog $ trans y):stack )
= --lemma 7
fst $ fst $ runProg (trans y) (sol, stack)
= --def of runNondet'
runNondet' y (sol, stack)
= --lemma's 2 and 8
sol ++ runNondet' Fail ([], []) ++ runNondet' y ([], stack)
```

- **case:** x = Ret a

```
runNondet' (x :| y) (sol, stack)
= --lemma 5
runNondet' (Ret a) (sol, (Prog $ trans y):stack)
=
fst $ fst $ runProg (emit a pop) (sol, (Prog $ trans y):stack)
= -- lemma 6
fst $ fst $ runProg pop (sol ++ [a], (Prog $ trans y):stack)
= --lemma 7
fst $ fst $ runProg (trans y) (sol ++ [a], stack)
= --def of runNondet'
runNondet' y (sol ++ [a], stack)
= --lemma's 3 and 8
sol ++ runNondet' (Ret a) [] [] ++ runNondet' y ([], stack)
```

- **case:** x = a :| b

```
runNondet' (x :| y) (sol, stack)
= --lemma 5
runNondet' (a :| b) (sol, (Prog $ trans y):stack)
= --Inductie
sol ++ runNondet' a ([], []) ++ runNondet' b ([], (Prog $ trans y):stack)
= --lemma 5
sol ++ runNondet' a ([], []) ++ runNondet' (b :| y) ([], stack)
= --Inductie
sol ++ runNondet' a ([], []) ++ runNondet' b [] [] ++ runNondet' y ([], stack)
```

```
= --Inductie
sol ++ runNondet' (a :| b) ([], []) ++ runNondet' y ([], stack)
```

The following lemma's are given without proof:

Lemma 5:

```
runNondet' (x :| y) (sol , stack)
=
runNondet' x (sol , (Prog $ trans y):stack)
```

Lemma 6:

```
runNondet' (emit x y) (sol , stack)
=
runNondet' y (sol ++ [x], stack)
```

Lemma 7:

```
runNondet' pop (sol , s:stack)
=
runNondet' s (sol, stack)
```

Lemma 8:

```
runNondet' x (sol , s:stack)
=
sol ++ runNondet' x ([], stack)
```

3 Results and Discussion

We intend to simulate backtrackable state using only state as an effect. We will first do this for concrete implementations of state and backtrackable state, in the style of the previous example. We will then provide a proof of such a simulation in the axiomatic style proposed by Gibbons and Hinze. It is our hope that working from the `State`-only characterisation of "backtrackable State" to an optimized backtracking algorithm will be a relatively short step and that this work will become useful for proving the correctness of such optimized backtracking algorithms.

References

1. Gibbons, J., Hinze, R.: Just do it: Simple monadic equational reasoning. Proceedings of the ACM SIGPLAN International Conference on Functional Programming, ICFP pp. 2–14 (2011). https://doi.org/10.1145/2034773.2034777
2. Hutton, G., Fulger, D.: Reasoning about effects: Seeing the wood through the trees. Tfp (2008), http://www.cs.nott.ac.uk/ gmh/effects.pdf
3. Kiselyov, O., Ishii, H.: Freer monads, more extensible effects. Haskell 2015 - Proceedings of the 8th ACM SIGPLAN Symposium on Haskell, co-located with ICFP 2015 pp. 94–105 (2015). https://doi.org/10.1145/2804302.2804319
4. Pauwels, K., Schrijvers, T., Mu, S.: Handling Local State with Global State pp. 18–44 (2019). https://doi.org/10.1007/978-3-030-33636-3_2
5. Wadler, P.: A critique of Abelsonand Sussmanor why calculating is better than scheming 1987 ACM SIGPLAN Notices, vol. 1 (1987)
6. Wadler, P.: Monads for functional programming. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics) **925**(August 1992), 24–52 (1995). https://doi.org/10.1007/978-3-662-02880-3_8