Concurrency and Nondeterminism using Effects and Handlers

Arthur Jacques, Supervisors: Casper Poulsen, Jaro Reinders

TU Delft

Background: Effects & Handlers

Algebraic effect and handlers separate effect descriptions from implementations, enhancing program modularity. Poulsen's approach [2] provides a basis for effect implementation. The images illustrate the free monad [3], state effect implementation, and a multi-effect example program.

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

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

```
incerr' = do
  s <- get'
  put' (s + 1)
  err' "foo"
```

Research task

My research task is to implement concurrency and nondeterminism using algebraic effects and handlers, such that they respect the laws given in "Modelling and analysis of communicating system" [1].

- Implement a representation for concurrency using new effects or combining existing ones.
- Implement a handler that will handle the nondeterminism inherent to concurrency.
- Prove the laws concerning concurrency from the book 'Modelling and Analysis of Communicating Systems' [1].
- Compare this implementation to existing literature and explore the possible combinations with other existing effects.
- Show some examples of useful programs using this implementation.

Methodology

The implementation extends code from Poulsen's blog post [2], using Haskell to create effects and handlers. Equational reasoning was used for law proofs, written in text files. The implementation and proofs are available in the project repository.

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. 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, as illustrated in the following figure.

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

x and y are the values returned by a and b. *par* a b is an interleaving of the effects of a and b, and returns the tuple (x,y).

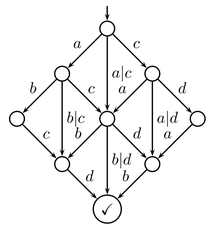
Nondeterminism

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

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

The Choose effect implements nondeterminism through a function taking a boolean as a parameter. $+$ uses that effect to implement the plus operator, which represents a division into two subprograms.

Nondeterminism in concurency



Concurrency comes with some inherent nondeterminism, coming from the order of interleaving of the actions of the program. This figure [1], shows that nondeterminism.

Implementation

```
-- function for running two programs concurrently (|||)
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)

-- function for running two programs concurrently, with the first one having priority
goesFirst :: Choose <: f => Free f a -> Free f b -> Free f (a, b)
goesFirst (Pure x) y = fmap (x,) y
goesFirst (Op x) y = Op (fmap (`par` y) x)
```

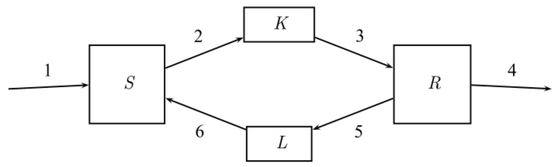
The function *par* works by using the *Choose* effect to non-deterministically decide which of the two programs to prioritize. This prioritization, which is the same as the leftmerge operation from the concurrency laws, is encapsulated in the function *goesFirst*, which calls *par* recursively with the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

Laws for Concurrency

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	$(x \mid y) \between z = x \mid (y \between z)$

This figure lists all the concurrency laws, and is taken from "Modelling and Analysis of communicating systems" [1]. We were able to prove that they hold for our interface, except for the ones concerning simultaneity, which is a possible future direction for this interface.

Application: ABP Model



The ABP model ensures reliable data transmission from sender (S) to receiver (R) over lossy channels. We implemented this model using our interface. We then proved its correctness as well as the validity of the model using equational reasoning. We also used handlers to execute the program whichever way we wanted, illustrating the versatility of effects and handlers.

Bibliography

References

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

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

[3] Wouter Swierstra. Data types à la carte. *Journal of Functional Programming*, 18(4), July 2008.