Linearity as an Effect

Hashan Punchihewa

Supervised by Dr. Nicolas Wu

22nd June 2020

Introduction

Types

$f = \texttt{"Hello"} * 5$

✗

Running Example: File Handles

```
main :: IO ()
main = do
  handle ← openFile "presentation.tex" ReadMode
  hPutStr handle "Fin"
```

✗

Two Different Views of Data

- Normal View = Data as an Abstract Entity
  - You can do whatever you want
- Linear View = Data as a Resource
  - There are rules that limit what you can do

Resources

- File Handles
- Network Sockets
- Currency
- Peripherals
- Arrays
- Qubits
- Energy
- Time

Computational Effects

- Resources are interacted with through computational effects
- Examples of effects: State and I/O
- Idea: Combine linearity with effects

Two Approaches to Effects in Haskell

- Monads (from Category Theory)
- Algebraic Effects (from Universal Algebra)

Algebraic Effects

Separation between Syntax & Semantics

- Syntax - Operations
- Semantics - Handlers

Algebraic Effects: Syntax

$$
\begin{aligned}
&\textbf{data } \mathit{FileEffect}\ x\ \textbf{where} \\
&\quad \mathit{Open} :: \mathit{String} \to \mathit{IOMode} \to \mathit{FileEffect}\ \mathit{Handle} \\
&\quad \mathit{Read} :: \mathit{Handle} \to \mathit{Int} \to \mathit{FileEffect}\ \mathit{String} \\
&\quad \mathit{Size} :: \mathit{Handle} \to \mathit{FileEffect}\ \mathit{Int} \\
&\quad \mathit{Write} :: \mathit{Handle} \to \mathit{String} \to \mathit{FileEffect}\ () \\
&\quad \mathit{Close} :: \mathit{Handle} \to \mathit{FileEffect}\ ()
\end{aligned}
$$

Algebraic Effects: Semantics

$$run :: Tree\ FileEffect\ a \to IO\ a$$

Algebraic Effects: Example

```
main :: IO ()
main = run do
  handle ← open "presentation.tex" ReadMode
  write handle "Fin"
```

✗

Parameterised Effects

- Combines linearity & algebraic effects.
- Associates each effect with a type-level state.
- They can depend on particular type-level state. (Precondition)
- They can modify the type-leve state. (Postcondition)

Type-Level State for File Handles

$$\begin{aligned}&\textbf{data } \mathit{IOModeT}\\&\quad = \mathit{ReadModeT} \mid \mathit{ReadWriteModeT} \mid \mathit{WriteModeT}\\&\textbf{type } \mathit{FileState} = \mathit{Maybe}\ \mathit{IOModeT}\end{aligned}$$

4 Possibilities:

- *Just ReadModeT*
- *Just ReadWriteModeT*
- *Just WriteModeT*
- *Nothing*

Type-Level State for File Handles

$$\begin{aligned}&\textbf{data } \mathit{IOModeT}\\&\quad = \mathit{ReadModeT} \mid \mathit{ReadWriteModeT} \mid \mathit{WriteModeT}\\&\textbf{type } \mathit{FileState} = \mathit{Maybe}\ \mathit{IOModeT}\\&\textbf{type } \mathit{FileEffectState} = [\mathit{FileState}]\end{aligned}$$

2 Different Meanings of State:

- Resource State, *FileState*
- Effect State, *FileEffectState*

File Handles Example

```
f = run do
  []
  handle ← open "presentation.tex" ReadMode
  [Just ReadModeT]
  close handle
  [Nothing]
  return ()
```

✓

File Handles Example

```
f = run do
  []
  handle ← open "presentation.tex" ReadMode
  [Just ReadModeT]
  close handle
  [Nothing]
  return ()
```

✓

File Handles Example

```
f = run do
  []
  handle ← open "presentation.tex" ReadMode
  [Just ReadModeT]
  close handle
  [Nothing]
  return ()
```

✓

File Handles Example

```
f = run do
  []
  handle ← open "presentation.tex" ReadMode
  [Just ReadModeT]
  handle' ← open "thesis.tex" WriteMode
  [Just ReadModeT, Just WriteModeT]
  close handle
  [Nothing, Just WriteModeT]
  close handle'
  [Nothing, Nothing]
  return ()
```

✓

File Handles Example

$$
\begin{aligned}
&f = \mathit{run}\ \mathbf{do} \\
&\quad [] \\
&\quad \mathit{handle} \leftarrow \mathit{open}\ \texttt{"presentation.tex"}\ \mathit{ReadMode} \\
&\quad [\mathit{Just}\ \mathit{ReadMode}T] \\
&\quad \mathit{handle}' \leftarrow \mathit{open}\ \texttt{"thesis.tex"}\ \mathit{WriteMode} \\
&\quad [\mathit{Just}\ \mathit{ReadMode}T, \mathit{Just}\ \mathit{WriteMode}T] \\
&\quad \mathit{close}\ \mathit{handle} \\
&\quad [\mathit{Nothing}, \mathit{Just}\ \mathit{WriteMode}T] \\
&\quad \mathit{close}\ \mathit{handle}' \\
&\quad [\mathit{Nothing}, \mathit{Nothing}] \\
&\quad \mathit{return}\ ()
\end{aligned}
$$

✓

File Handles Example

```
f = run do
  []
  handle ← open "presentation.tex" ReadMode
  [Just ReadModeT]
  handle' ← open "thesis.tex" WriteMode
  [Just ReadModeT, Just WriteModeT]
  close handle
  [Nothing, Just WriteModeT]
  close handle'
  [Nothing, Nothing]
  return ()
```

✓

File Handles Example

```
f = run do
  []
  handle ← open "presentation.tex" ReadMode
  [Just ReadModeT]
  handle′ ← open "thesis.tex" WriteMode
  [Just ReadModeT, Just WriteModeT]
  close handle
  [Nothing, Just WriteModeT]
  close handle′
  [Nothing, Nothing]
  return ()
```

✓

File Handles Example

f = *run* **do**
  []
  handle ← *open* `"presentation.tex"` *ReadMode*
  [*Just ReadModeT*]
  handle′ ← *open* `"thesis.tex"` *WriteMode*
  [*Just ReadModeT*, *Just WriteModeT*]
  close handle
  [*Nothing*, *Just WriteModeT*]
  close handle′
  [*Nothing*, *Nothing*]
  return ()

✓

File Handles Example

```
f = run do
    handle ← open "presentation.tex" ReadMode
    [Just ReadModeT]
    write handle "Fin"
    return ()
```

✗

File Handles Example

```
f = run do
    handle ← open "presentation.tex" WriteMode
    [Just WriteModeT]
    write handle "Fin"
    return ()
```

✗

File Handles Example

```
f = run do
  handle ← open "presentation.tex" WriteMode
  [Just WriteModeT]
  write handle "Fin"
  close handle
  [Nothing]
  return ()
```

✓

Implementation

Type-Level Computation

- GHC has rich type-level capabilities
- Extensions such as *DataKinds*, *TypeFamilies*, *PolyKinds*, etc.

Lists of Resource States

$$\textbf{type } \mathit{FileState} = \mathit{Maybe}\ \mathit{IOModeT}$$
$$\textbf{type } \mathit{FileEffectState} = [\mathit{FileState}]$$

Type-families to work with lists of resource states:

- *Append*
- *Replace*
- *Lookup*

File Handles Example

```
f = run do
  []
  handle ← open "presentation.tex" WriteMode
  [Just WriteModeT]
  write handle "Fin"
  close handle
  [Nothing]
  return ()
```

File Handles Example

```
f = run do
  []
  handle ← open "presentation.tex" WriteMode
  [Just WriteModeT]
  write handle "Fin"
  close handle
  [Nothing]
  return ()
```

File Handles Example

```
f = run do
  []
  handle ← open "presentation.tex" WriteMode
  [Just WriteModeT]
  write handle "Fin"
  close handle
  [Nothing]
  return ()
```

File Handles Example

```
f = run do
  []
  handle ← open "presentation.tex" WriteMode
  [Just WriteModeT]
  write handle "Fin"
  close handle
  [Nothing]
  return ()
```

Encoding of Operations

```
data FileEffect p q x where
  Open
    :: String → IOMode m
    → FileEffect
        p
        (Append p (Just m))
        (Handle (Len p))
```

Encoding of Operations

```
data FileEffect p q x where
  Open
    :: String → IOMode m
    → FileEffect
        p
        (Append p (Just m))
        (Handle (Len p))
```

Encoding of Operations

```
data FileEffect p q x where
  Open
    :: String → IOMode m
    → FileEffect
        p
        ( Append p (Just m) )
        (Handle (Len p))
```

Encoding of Operations

```
data FileEffect p q x where
  Open
    :: String → IOMode m
    → FileEffect
        p
        (Append p (Just m))
        ( Handle (Len p) )
```

Encoding of Operations

```
data FileEffect p q x where
  Open
    :: String → IOMode m
    → FileEffect
        p
        (Append p (Just m))
        (Handle (Len p))
```

Encoding of Operations

```
data FileEffect p q x where
  Open
    :: String → IOMode m
    → FileEffect
        p
        (Append p (Just m))
        (Handle (Len p))
```

Indexed Types

```
type IOMode :: IOModeT → Type
data IOMode where
  ReadMode      :: IOMode ReadModeT
  WriteMode     :: IOMode WriteModeT
  ReadWriteMode :: IOMode ReadWriteModeT
```

Encoding of Operations

```
data FileEffect p q x where
  Open
    :: String → IOMode m
    → FileEffect
        p
        (Append p (Just m))
        ( Handle (Len p) )
```

Indexed Types

$$\textbf{data}\ \mathit{Nat} = \mathit{Z} \mid \mathit{S}\ \mathit{Nat}$$

$$\textbf{type}\ \mathit{Handle} :: \mathit{Nat} \rightarrow \mathit{Type}$$
$$\textbf{data}\ \mathit{Handle}\ n = \ldots$$

Encoding of Operations

$$
\begin{aligned}
&\textbf{data}\ \mathit{FileEffect}\ p\ q\ x\ \textbf{where}\\
&\quad \mathit{Write}\\
&\qquad :: (\mathit{Lookup}\ p\ n \sim \mathit{Just}\ m, \mathit{CanWrite}\ m)\\
&\qquad \Rightarrow \mathit{Handle}\ n\\
&\qquad \rightarrow \mathit{String}\\
&\qquad \rightarrow \mathit{FileEffect}\\
&\qquad\qquad p\\
&\qquad\qquad p\\
&\qquad\qquad ()
\end{aligned}
$$

Encoding of Operations

```
data FileEffect p q x where
  Write
    :: (Lookup p n ~ Just m, CanWrite m)
    => Handle n
    -> String
    -> FileEffect
         p
         p
         ()
```

Encoding of Operations

```
data FileEffect p q x where
  Write
    :: (Lookup p n ∼ Just m, CanWrite m)
    ⇒ Handle n
    → String
    → FileEffect
        p
        p
        ()
```

Encoding of Operations

```
data FileEffect p q x where
  Write
    :: (Lookup p n ~ Just m, CanWrite m)
    => Handle n
    -> String
    -> FileEffect
         p
         p
         ()
```

Composition of Effects

- Can combine effects together.
- Effects can be for different resources.
- 3 different type-level states:
  - Combined State
  - Effect State
  - Resource State

Composition of Effects

- List of effects
  $[FileEffect, SocketEffect]$
- List of effect states
  $[[Just\ WriteModeT], [Just\ Listening]]$
- If *xs* is a list of effects, *Coproduct xs* is one effect from *xs*

Constraints for Lists of Effect States

- *Contains c i x*
- *Evolve c i x d*

Examples:

- *Contains* [[*Just WriteModeT*]] *Z* [*Just WriteModeT*]
- *Evolve* [[*Just WriteModeT*]] *Z* [*Nothing*] [[*Nothing*]]

Constraints for Lists of Effect States

- *Contains c i x*
- *Evolve c i x d*

Examples:

- *Contains* [[*Just WriteModeT*]] *Z* [*Just WriteModeT*]
- *Evolve* [[*Just WriteModeT*]] *Z* [*Nothing*] [[*Nothing*]]

Constraints for Lists of Effect States

- *Contains c i x*
- *Evolve c i x d*

Examples:

- *Contains* [[*Just WriteModeT*]] *Z* [*Just WriteModeT*]
- *Evolve* [[*Just WriteModeT*]] *Z* [*Nothing*] [[*Nothing*]]

Constraints for Lists of Effect States

- *Contains c i x*
- *Evolve c i x d*

Examples:

- *Contains* [[*Just WriteModeT*]] *Z* [*Just WriteModeT*]
- *Evolve* [[*Just WriteModeT*]] *Z* [*Nothing*] [[*Nothing*]]

Constraints for Lists of Effect States

- *Contains c i x*
- *Evolve c i x d*

Examples:

- *Contains* [[*Just WriteModeT*]] *Z* [*Just WriteModeT*]
- *Evolve* [[*Just WriteModeT*]] *Z* [*Nothing*] [[*Nothing*]]

Constraints for Lists of Effect States

- *Contains c i x*
- *Evolve c i x d*

Examples:

- *Contains* [[*Just WriteModeT*]] *Z* [*Just WriteModeT*]
- *Evolve* [[*Just WriteModeT*]] *Z* [*Nothing*] [[*Nothing*]]

Type-Checker Plugin

```
f = run do
  handle ← open "presentation.tex" WriteMode
  write handle "Fin"
  close handle
  return ()
```

Type-Checker Plugin

```
f = do
  handle ← open "presentation.tex" WriteMode
  write handle "Fin"
  close handle
  return ()
```

Type-Checker Plugin

```
f = do
  handle ← open "presentation.tex" WriteMode
  write handle "Fin"
  close handle
  return ()
```

Haskell gets confused, because it doesn't know:

- Which effects there are.
- What the effect states for these effects are.

Type-Checker Plugin

$$
\begin{aligned}
f :: (&\mathit{CanWrite}\ m, \mathit{MemberAux}\ (\mathit{FindEffect}\ \mathit{FileEffect}\ u)\ \mathit{FileEffect}\ u,\\
&\mathit{Contains}\ (\mathit{FindEffect}\ \mathit{FileEffect}\ u)\ p\ c,\\
&\mathit{Contains}\ (\mathit{FindEffect}\ \mathit{FileEffect}\ u)\ q\ j,\\
&\mathit{Contains}\ (\mathit{FindEffect}\ \mathit{FileEffect}\ u)\ p1\ j1,\\
&\mathit{Evolve}\\
&\quad c\ (\mathit{FindEffect}\ \mathit{FileEffect}\ u)\ (\mathit{Append}\ p\ (\mathit{Just}\ \mathit{WriteModeT}))\ j,\\
&\mathit{Evolve}\ j\ (\mathit{FindEffect}\ \mathit{FileEffect}\ u)\ q\ j1,\\
&\mathit{Evolve}\\
&\quad j1\ (\mathit{FindEffect}\ \mathit{FileEffect}\ u)\ (\mathit{Replace}\ p1\ (\mathit{Len}\ p)\ \mathit{Nothing})\ d,\\
&\mathit{Lookup}\ p1\ (\mathit{Len}\ p) \sim \mathit{Just}\ m1, \mathit{Lookup}\ q\ (\mathit{Len}\ p) \sim \mathit{Just}\ m) \Rightarrow\\
&\mathit{ITree}\ u\ c\ d\ ()
\end{aligned}
$$

Type-Checker Plugin

```
f
  :: (Member FileEffect u a c i d,
      Evolve c (FindEffect FileEffect u) (Append a Nothing) d)
  ⇒ ITree u c d ()
```

Type-Checker Plugin

- Gets list of given and wanted constraints
- Converts constraints into special representation
- Runs fixed-point algorithm to simplfiy
- Updates Haskell with simplified constraints

Scoped Operations

```
f = run do
    ...
    state
    isolate do
        []
        handle ← open "presentation.txt" ReadMode
        [Just ReadModeT]
        close handle
        [Nothing]
        return ()
    state
    ...
```

Scoped Operations

```
f = run do
  ...
  state
  isolate do
    []
    handle ← open "presentation.txt" ReadMode
    [Just ReadModeT]
    close handle
    [Nothing]
    return ()
  state
  ...
```

Scoped Operations

```
f = run do
  ...
  state
  isolate do
    []
    handle ← open "presentation.txt" ReadMode
    [Just ReadModeT]
    close handle
    [Nothing]
    return ()
  state
  ...
```

Applications

Session Types

- Communication channels between two processes
- Statically guarantee, they adhere to the same protocol
- Advance features:
  - Session delegation
  - Recursive session channels

Session Types

Operations: *create*, *close*, *send*, *recv*, *chooseLeft*, *chooseRight*, *sendChannel*, *recvChannel*, *rec*, *recCall*
Scoped Operations: *fork*, *offer*

```
client channel = do
  v ← rec channel
  offer channel
    (do
      k ← recv channel
      injectIO (putStrLn (show k))
      recCall channel v
      client channel
    )
    (do
      injectIO (putStrLn "Goodbye!")
      close channel
    )
```

```
server channel = do
  v ← rec channel
  j ← injectIO (getStdRandom (randomR (1, 30)))
  if j ⩾ 3 then do
    chooseLeft channel
    send channel j
    recCall channel v
    server channel
  else do
    chooseRight channel
    close channel
  return ()
```

```
example = run do
  channel ← create
  fork (client channel)
  server channel
```

Mutable Arrays

- Multiple threads with access to the same array
- Statically guarantee no data races
- Example: Concurrent Quicksort

Mutable Arrays

Operations: *alloc*, *read*, *write*, *length*, *slice*, *join*, *wait*
Scoped Operations: *fork*

Conclusion

- Linearity means *Data as a Resource*
- The goal of the project was to build a framework for encoding linearity in Haskell as a computational effect.
- Several examples were built to evaluate the project.