Strong Types and Pure Functions

Enforcing control of side effects in interfaces

Duncan Coutts

Well-Typed

Functional Programming eXchange 2009

Before we start. . . syntax

```
let getInterestingNumber
  (cities:Map<string,int>) (population:Map<int,int>)
  (cityName:string) :int Option =
    maybe{
      let! zipCode        = cities.TryFind ciyName
      let! cityPopulation = population.TryFind zipCode
      return cityPopulation * 100 / TOTAL_POPULATION }
```

$$\mathit{getInterestingNumber} :: \mathit{Map\ String\ Int} \rightarrow \mathit{Map\ Int\ Int} \rightarrow \mathit{String} \rightarrow \mathit{Maybe\ Int}$$

$$\begin{aligned}
&\mathit{getInterestingNumber\ cities\ population\ cityName} = \textbf{do} \\
&\quad \mathit{zipCode} \leftarrow \mathit{Map.lookup\ citName\ cities} \\
&\quad \mathit{cityPopulation} \leftarrow \mathit{Map.lookup\ zipCode\ population} \\
&\quad \mathit{return}\ (\mathit{cityPopulation} * 100\ /\ \mathit{totalPopulation})
\end{aligned}$$

Before we start. . . syntax

```
let getInterestingNumber
  (cities:Map<string,int>) (population:Map<int,int>)
  (cityName:string) :int Option =
    maybe{
      let! zipCode        = cities.TryFind ciyName
      let! cityPopulation = population.TryFind zipCode
      return cityPopulation * 100 / TOTAL_POPULATION }
```

$$getInterestingNumber :: Map\ String\ Int \rightarrow Map\ Int\ Int \rightarrow String \rightarrow Maybe\ Int$$

$$getInterestingNumber\ cities\ population\ cityName = \mathbf{do}$$
$$\quad zipCode \leftarrow Map.lookup\ citName\ cities$$
$$\quad cityPopulation \leftarrow Map.lookup\ zipCode\ population$$
$$\quad return\ (cityPopulation * 100\ /\ totalPopulation)$$

Why limit side effects?

- Lots of reasons to reduce side effects in general
- Some tasks require side effects
- Sometimes we wish to guarantee the absence of certain side effects but still allow other side effects

Where guaranteed limits are useful

Frameworks with callbacks (don't trust external code)

- May require promises from callbacks, e.g.
  - no hidden shared state
  - transaction safe
  - disciplined use of expensive resources
- May require visibility into callbacks, e.g.
  - synchronisation mediated by the framework
  - dynamic checks

Internal interfaces in a system (don't trust your own code!)

Our goal

We want to design interfaces so that the implementations

- may not use certain effects
- may use some effects

We want strong guarantees

How to enforce control of effects?

- Restricting particular side effects is hard
- Build up, not down
  - start with no side effects
  - add just the effects you wish to allow

Types tell us what functions can do

In a pure FP language, functions have no effects at all

$$f :: A \to B$$

Effectful actions can do anything

$$f :: A \to IO\ B$$

Types let us “tag” the effects

Simple case

Return pure descriptions of what to do

Example: serving HTTP requests

$$\textbf{type}\ \mathit{Handler} = \mathit{Request} \rightarrow \mathit{Response}$$

But what about

- talking to a DB
- expensive resources
- waiting on other servers

Restricted actions

We have both extremes

$$f :: A \rightarrow \quad B$$
$$f :: A \rightarrow IO\ B$$

Want to build restricted actions

$$f :: A \rightarrow Sandbox\ B$$

Building sandboxes

Want restricted actions

$$f :: A \rightarrow Sandbox\ B$$

- Build pure description of effectful actions
- Must be able to describe complex compound actions
- Trusted interpreter performs the effects

$$interpret :: Sandbox\ a \rightarrow IO\ a$$

Describing effects: monads

This is not a tutorial on monads!

Monads are how we give pure descriptions of actions with side effects

What effects we can describe

Monad pattern can describe

- Mutable state
- Input/Output
- Exceptions
- Concurrency
- Co-routines
- Continuations
- Backtracking
- Non-determinism

IO monad already contains all of these

We get to pick and mix when we build our own

What effects we can describe

Monad pattern can describe

- Mutable state
- Input/Output
- Exceptions
- Concurrency
- Co-routines
- Continuations
- Backtracking
- Non-determinism

IO monad already contains all of these

We get to pick and mix when we build our own

Example: hooks in a build system

Build system with hooks for custom actions: configure, build, etc.

Many rules custom actions should respect, e.g.

- Should not modify files outside of build dir
- Should respect installation prefix

Need some introspection into custom actions

- Debugging, logging
- Track build dependencies

Must be able to run complex custom actions

Interface

We'll define a "*Sh*" shell monad and use it in the interface

$$
\begin{aligned}
&\textbf{data}\ \mathit{BuildHooks} = \mathit{BuildHooks}\ \{ \\
&\quad \mathit{configureHook} :: \mathit{PkgDescription} \\
&\qquad\qquad \rightarrow \mathit{ConfigureFlags} \\
&\qquad\qquad \rightarrow \mathit{Sh}\ \mathit{BuildInfo}, \\
&\quad \mathit{buildHook} :: \mathit{PkgDescription} \\
&\qquad\qquad \rightarrow \mathit{BuildInfo} \\
&\qquad\qquad \rightarrow \mathit{BuildFlags} \\
&\qquad\qquad \rightarrow \mathit{Sh}\ (), \\
&\qquad \ldots \\
&\}
\end{aligned}
$$

Pure action description

Explicit, pure description of actions

$$
\begin{aligned}
&\textbf{data}\ \mathit{Action}\ a = \\
&\quad \mathit{Stop}\ a \\
&\quad \mid \mathit{Fail}\ \ \mathit{String} \\
&\quad \mid \mathit{ReadFile}\ \mathit{FilePath}\ \ (\mathit{String} \rightarrow \mathit{Action}\ a) \\
&\quad \mid \mathit{WriteFile}\ \mathit{FilePath}\ \mathit{String}\ (\mathit{Action}\ a)
\end{aligned}
$$

Last field in each case is the *next* action

Types and pure functions guarantee no unexpected hidden effects here

Example action description

Action data structures look like

$$
\begin{aligned}
&\mathit{copyFile} :: \mathit{FilePath} \to \mathit{FilePath} \\
&\qquad \to \mathit{Action}\ a \to \mathit{Action}\ a \\
&\mathit{copyFile}\ \mathit{from}\ \mathit{to}\ \mathit{next} = \\
&\quad \mathit{ReadFile}\ \mathit{from}\ (\lambda \mathit{content} \to \\
&\qquad \mathit{WriteFile}\ \mathit{to}\ \mathit{content}\ \mathit{next})
\end{aligned}
$$

- Data structure contains lots of (pure) functions
- “next” parameter appears in lots of places

We do not want to have to write code like this!

Trusted interpreter

Trusted interpreter performs the effects

```
interpret :: Action a → IO a
interpret (Stop result) = return result
interpret (Fail msg)    = fail msg
interpret (ReadFile file next) = do
  content ← System.IO.readFile file
  interpret (next content)
interpret (WriteFile file content next) = do
  System.IO.writeFile file content
  interpret next
```

Example: *interpret*

Actually perform the action we described

interpret (*copyFile* `"a.txt" "b.txt"` (*Stop* ()))

Pure simulator

```
simulate :: [String] → Action a → [String]
simulate log (Stop _)    = reverse log
simulate log (Fail msg) =
  let entry = "failed: " ++ msg
  in reverse (entry : log)
simulate log (ReadFile file next) =
  let content = "(contents of " ++ file ++ ")"
      entry   = "read " ++ file
  in simulate (entry : log) (next content)
simulate log (WriteFile file content next) =
  let entry = "write " ++ file ++ "  " ++ content
  in simulate (entry : log) next
```

Building *Action* descriptions

We want to build *Action* descriptions in a nicer way

Would like to write

$$
\begin{aligned}
&\mathit{copyFile\ from\ to} = \mathbf{do} \\
&\quad \mathit{content} \leftarrow \mathit{readFile\ from} \\
&\quad \mathit{writeFile\ to\ content}
\end{aligned}
$$

Uses the monad syntax

Shell monad

Definition of shell monad

$$\begin{aligned}
&\textbf{newtype } \mathit{Sh}\ a = \\
&\quad \mathit{MkShell}\ (\forall r.\ (a \to \mathit{Action}\ r) \to \mathit{Action}\ r) \\
&\mathit{unShell} :: \mathit{Sh}\ a \to (a \to \mathit{Action}\ r) \to \mathit{Action}\ r \\
&\mathit{unShell}\ (\mathit{MkShell}\ sh)\ \mathit{next} = sh\ \mathit{next}
\end{aligned}$$

Standard definition of a “continuation monad”

Shell monad

Standard continuation monad instance

$$
\begin{aligned}
&\textbf{instance } \mathit{Monad\ Sh} \textbf{ where} \\
&\quad \mathit{return\ a} = \mathit{MkShell}\ (\lambda \mathit{next} \rightarrow \mathit{next\ a}) \\
&\quad m \gg\!= f \ = \mathit{MkShell}\ (\lambda \mathit{next} \rightarrow \\
&\qquad\qquad \mathit{unShell\ m}\ (\lambda a \rightarrow \mathit{unShell}\ (f\ a)\ \mathit{next})) \\
&\quad \mathit{fail\ msg} = \mathit{MkShell}\ (\lambda \_ \rightarrow \mathit{Fail\ msg})
\end{aligned}
$$

Hides all the plumbing of the “next” parameter

Running shell actions

$$asAction :: Sh\ a \rightarrow Action\ a$$
$$asAction\ sh = unShell\ sh\ Stop$$

$$runShell :: Sh\ a \rightarrow IO\ a$$
$$runShell = interpret \circ asAction$$

$$debugShell :: Sh\ a \rightarrow [String]$$
$$debugShell = simulate\ [\,] \circ asAction$$

- Use *Sh* monad to construct *Action* data structure
- Interpret *Action* data structure

Shell monad primitive actions

Definition of primitive effectful *Sh* actions

$$
\begin{aligned}
&\mathit{readFile} :: \mathit{FilePath} \to \mathit{Sh}\ \mathit{String} \\
&\mathit{readFile}\ \mathit{file} = \\
&\quad \mathit{MkShell}\ (\lambda \mathit{next} \to \mathit{ReadFile}\ \mathit{file}\ \mathit{next}) \\
&\mathit{writeFile} :: \mathit{FilePath} \to \mathit{String} \to \mathit{Sh}\ () \\
&\mathit{writeFile}\ \mathit{file}\ \mathit{content} = \\
&\quad \mathit{MkShell}\ (\lambda \mathit{next} \to \mathit{WriteFile}\ \mathit{file}\ \mathit{content}\ (\mathit{next}\ ()))
\end{aligned}
$$

Client code will not use the *Action* data structure directly

Shell examples

Can now write the *copyFile* example

```
copyFile :: FilePath → FilePath → Sh ()
copyFile from to = do
    content ← readFile from
    writeFile to content
```

Use standard monad functions to help make more complex actions

```
cat :: [FilePath] → FilePath → Sh ()
cat files target = do
    contents ← mapM readFile files
    writeFile target (concat contents)
```

Introspection

Now we have visibility into key actions

- Can do various dynamic checks
- Can change the behaviour
- Just change the interpreter!
  (or add another interpreter)

Example: Web request handling

Web app framework

- Framework provides state/DB service
- Otherwise require page handlers to be stateless (so they can safely be run concurrently)
- Could provide read-only access to static files
  - Use dynamic checks to guarantee only access to files inside web root

Example: DB connections

Application plugin that is permitted access to DB

- Can provide safe access to DB connection
- Could permit only queries
- Could enforce disciplined resource control

$$\begin{aligned} \mathit{withResource} &:: \mathit{Options} \\ &\rightarrow (\mathit{Resource} \rightarrow \mathit{DbPlugin}\ a) \\ &\rightarrow \mathit{DbPlugin}\ a \end{aligned}$$

block scoped

$$\mathit{withResource}\ \mathit{opts}\ \$\ \lambda \mathit{res} \rightarrow \textbf{do}$$

$$\ldots$$

Related approaches

Using an explicit data structure to represent actions is a "deep embedding"

$$asAction :: Sh\ a \longrightarrow Action\ a$$
$$interpret :: Action\ a \longrightarrow IO\ a$$

Can also use "shallow embedding": fuses interpreter with definition of monad and primitive actions

$$runShell :: Sh\ a \longrightarrow IO\ a$$

Concerns: is it too slow?

- High performance web servers built with this technique
- Co-routines between compiled code
- Overhead depends on granularity of operations
- Lower overhead possible with shallow embedding

Concerns: is it extensible?

Might need several types of restricted action, each extending the previous

- Use type classes
- Easier with shallow embedding

Next steps

Concepts

- Functors, Monads
- MTL — Monad Transformer Library

Books

- “Programming in Haskell” — best quickstart
- “Real World Haskell” — good coverage, very practical

Summary

- Continuum of custom restricted effects

$$
\begin{array}{l}
:: A \to \quad B \\
\vdots \\
:: A \to Sh\ B \\
\vdots \\
:: A \to IO\ B
\end{array}
$$

- Pick the effects you want to allow
- Design your interface
- Types enforce the interface contract