Algebraic Effects in practice (and in theory too)

Zeimer

8 November 2018

Referential transparency in natural languages - definitions

- Referential transparency is a concept coming from analytic philosophy
- A referent of a phrase is the thing to which that phrase refers. For example, the phrase "The capital of Scotland" refers to Edinburgh.
- A context is a part of a phrase with a hole, for example "_ is a beautiful city".
- A context is referentially transparent when we can substitute terms with the same referent for each other without changing the meaning of the sentence.
- A context is referentially opaque if the above is not the case.

Referential transparency in natural languages - examples

- Consider the context "_ is a beautiful city".
- The sentences "The capital of Scotland is a beautiful city" and "Edinburgh is a beautiful city" have the same meaning. This means that this context is referentially transparent.
- Consider the context "_ has been the capital of Scotland since 1999".
- The sentences "Edinburgh has been the capital of Scotland since 1999" and "The capital of Scotland has been the capital of Scotland since 1999" mean something different, so this context is referentially opaque.
- Thus referential opacity is a form of context dependence, a dependence on the outside world.

Referential transparency in programming languages

- The concept was borrowed to programming languages: a name is referentially transparent if

Referential transparency - examples

- `let x = unsafePerformIO getCPUTime in x == x`
- `unsafePerformIO getCPUTime == unsafePerformIO getCPUTime`

Values and computations

- Not all effects are monads. Which ones are and which ones aren't?

Value or computation?

- Consider the type $E \to A$.
- We can consider elements of this type to be values. They are pure functions, after all.
- We can, however, also see them as effectful computations of type A that depend on some external environment E.
- Which point of view is better is ad hoc. Not many people consider pure functions $E \to A$ effectful, and not many people consider $IO\ a$ pure in Haskell.

Examples of effects

- Nondeterminism and randomness
- Mutable state, reading configuration and logging
- Exceptions, partiality and errors
- Continuations - callCC, shift, reset and whatnot
- Input and output
- Nontermination
- Allocating memory
- Threads
- Asynchronicity

The same effect or different?

- Consider the effect of performing some input/output operations.
- Many things qualify - reading and writing to standard input/output, manipulating files, connecting to a local network or the Internet, manipulating SQL/NoSQL database, running a HTTP server, reading data from microphone, camera, joystick, network card...
- Should these be considered different effects or parts of the IO effect?
- If your language is too weak, then they are the same.
- If your language is strong enough, you may consider them the same or different depending on what you need (e.g. security or ease of implementation).

An effect or not?

- Consider an operation `sleep(time)`. Can calling it be regarded as an effectful behaviour?

Where do effects come from?

- What effects are available in a language depends on the design (semantics) of that language.
- In theory, any Turing-complete language can express any effect (by implementing a compiler or interpreter of a language which supports these effects).
- In practice, some languages can express more effects than others or they can do it more naturally/easily.

Where do effects come from: Java

- Nontermination: unrestricted while loop.
- Input and output: library functions for performing IO.
- Partiality: objects of any class can be null.
- Exceptions: built-in exceptions mechanism.
- Mutable state: assignment.
- Randomness: implementable (impure pseudorandom generators).

Where do effects come from: Haskell

- Nontermination: unrestricted recursive calls.
- Input and output: built-in IO type and functions for using it.
- Partiality: can be implemented using algebraic data types (the type Maybe).
- Exceptions: the `error` function (note that this is something different than in Java). Java styled exceptions can be implemented using continuations.
- Mutable state: can be implemented as a monad with the type $s \longrightarrow (a, s)$
- Randomness: implementable using pure pseudorandom generators.

Where do effects come from: Coq

- Coq is a theorem prover and pure, total function programming language: https://coq.inria.fr/
- Nontermination: impossible because only structural recursion is allowed.
- Input and output: impossible (no built-in IO).
- Exceptions: no Java style exceptions, but can be simulated using continuations.
- Partiality, mutable state, randomness: can be implemented like in Haskell.

What you can and must express

- Types can be used to tell what effects a computation can have.
- In practice, types can tell us more (Haskell) or less (Java) and this is very useful.
- In theory, however, in different languages the same types (like 32-bit integers) can mean different things depending on what effects are available.
- Let's write $A!\{e_1, \ldots, e_n\}$ for a computation which returns a value of type A and can have effects $e_1, \ldots, e_n$.

The meaning of types: Java

- The meaning of a function $A \to B$ where B is a primitive type can be seen as $A \to B!\{\bot, \mathrm{IO}, \mathrm{Unchecked}, \ldots\}$, because this function may loop, perform IO, throw unchecked exceptions. The three dots signify that it can do even more, i. e. return a random value. But we know that it can't return null or a list of Bs.
- If B is not a primitive type, then $A \to B$ means $A \to B!\{\bot, \mathrm{IO}, \mathrm{Unchecked}, \mathrm{Null}, \ldots\}$ - now the result can be null.
- If there's a checked exception E in the signature of the function, then the type $A \to B$ can be interpreted as $A \to B!\{\bot, \mathrm{IO}, \mathrm{Unchecked}, \mathrm{Null}, E, \ldots\}$, where E signifies an effect of throwing the checked exception.

The meaning of types: Haskell

- Surprisingly, Haskell is not very far from Java.
- A Haskell function of type $A \to B$ can be interpreted as $A \to B!\{\bot, \mathrm{IO}, \mathrm{Error}, \ldots\}$, because it may loop, perform IO (through functions like `unsafePerformIO`), result in an error (like when calling `head` on an empty list) and many more (because we can do a lot with unsafe IO).
- A function of type $A \to M(B)$, where M is some monad, can be interpreted as $A \to B!\{\bot, \mathrm{IO}, \mathrm{Error}, M, \ldots\}$, where M signifies the effect of this monad (e.g. nondeterminism for the list monad).

The meaning of types: Coq

- Coq is different (otherwise I wouldn't have included it in the examples).
- A Coq function of type $A \longrightarrow B$ means $A \longrightarrow B!\{\}$, because it can't have any effect - it must terminate, can't perform IO or exceptions, can't return null etc.
- A function of type $A \longrightarrow M(B)$ for some monad M means $A \longrightarrow B!\{M\}$.

How effects are managed: Java

- Nontermination: you can't do anything about it.
- Partiality: null checks everywhere.
- Exceptions: `throw`, `catch`, `finally`. Checked exceptions appear in function signatures.
- Other effects: because Java is not very effect-aware, you can only manage them ad hoc by using them wisely.

How effects are handled: Haskell

- Nontermination: you can't do anything about it.
- Input and output: the IO monad, but you can't do anything when it's done through `unsafePerformIO`.
- Exceptions (the `error` function): this can be caught, but it's a bad idea. Better avoid this effect.
- Other effects: monads (and applicatives too).

How effects are handled: Coq

- The only effects you have come from monads/applicatives, so you use these to handle them.

●

Bibliography

- Referential transparencyhttps://stackoverflow.com/questions/210835/what-is-referential-transparency