Algebraic Effects in practice (and in theory too)

Zeimer

22 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 - definitions

- The concept was borrowed to programming languages, but since programmers rarely talk about contexts, it evolved and refers to functions.
- A function is referentially transparent iff its output depends only on its input (or equivalently: iff called many times with the same arguments it gives the same result).
- A programming language is referentially transparent iff all functions definable in this language are referentially transparent.

Referential transparency in programming languages - examples

- Java is not referentially transparent, because a function call like `System.currentTimeMillis()` produces a different value each time.
- Haskell is also not referentially transparent, because of a similar problem: `unsafePerformIO getCPUTime` returns a different value each time it's called.
- Coq is a theorem prover and pure, total function programming language. It is referentially transparent - you can't get the current time nor anything like that.
- As we see, the lack of referential transparency in programming languages has a similar cause to that from natural languages - context dependence, which means access to some external state.

Purity - definitions

- A concept related to referential transparency is purity. A function is **pure** if it has no **side effects**, which are understood as (read and/or write) access to some external state.
- This can be generalized a bit to expressions: an expressions is **pure** if evaluating it produces no side effects.
- Accordingly a programming language is pure (or purely functional) if all it's functions/expressions are pure.
- Note: sometimes the definition is expanded so that nontermination is considered impure.

Purity - examples

- Since purity implies referential transparency, neither Java nor Haskell are pure.
- But Coq is pure - this is because it's a theorem prover meant for doing mathematics. Yes, you can't do any IO in it, but you can prove theorems! It's also pure in the stronger sense that it doesn't allow nontermination - thus it's not Turing complete.

Referential transparency and purity are not absolute

- Both referential transparency and purity are very strong properties for a language to posses. Since barely any languages have them (except Coq, of course), it is more useful to consider languages more or less referentially transparent/pure depending on what they enforce and encourage.
- In Haskell it's easier to write referentially transparent functions. It is encouraged, it is enforced by the type system and getting around it requires some hackery. Therefore we can say that Haskell is more referentially transparent than Java.
- The same goes for purity - in Java even the simplest "Hello world" program is impure. In Haskell, in contrast, most of the IO is pure (why and how we will learn soon). Therefore, Haskell is more pure than Java.

Why bother?

- Why should we care about about referential transparency and purity?
- They allow for certain optimizations.
- Enforcing them prevents some silly mistakes.
- They facilitate reasoning, especially equational reasoning.
- They are compositional, which allows decomposing big architectures into simple components.
- Therefore it would be a good idea to have some conceptual, formal and technological tool to make referential transparency and purity easier.

Values and computations - definitions

- This suggests dividing (at least conceptually) all stuff out there into values and computations.
- A **value** is something that just is there. Value is a being like 2 or something that reduces to a being, like $1 + 1$.
- A **computation** is a process that does something. Computation is doing, like connecting to the Internet or writing to the database.

Values and computations - do be do be do

- Now here's the trick to get free referential transparency and purity: reification.
- Instead of doing something (performing a computation), we can create an object (a value) which tells us how to perform that computation.
- Instead of performing many computations, we can put together the objects that represent them.
- In the meantime we can mess with these objects, change them and interpret/transform them into something else without introducing any impurity/opacity.
- Finally, we can put that big object representing all our (remaining) impure computations into a single point of contact with the outside world (usually the `main` function. This is why most of Haskell IO is pure and referentially transparent.

Values and computations - example

- Consider the type $E \to A$.
- We can think that elements of this are values. They are pure functions, after all.
- We can, however, also see them as computations of type A that depend on some external environment E.
- This is the essence of the above trick that applies to any computation whatsoever.

Effects - definition

- Are we done? Not really. We said we want to put computations (or rather, their descriptions) together and mess with them in the meantime.
- For this, we have to dissect computations into parts. A computation like "add n to m and print the result to the screen" has a pure part (addition) and an impure part (printing).
- By an effect **effect** I will mean an impure part of a computation.

Effects - examples

- Nondeterminism and randomness
- Reading configuration and logging
- Exceptions, partiality and errors
- Continuations - callCC, shift, reset and whatnot
- Input and output
- Memory management (allocation and freeing)
- Asynchronicity, concurrency, threads
- Since effects by definitions are impure, most of the above boil down to access to global state. This raises the question: when are effects the same/different?

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?

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

The meaning of types

- It is usually said that types classify values. But since we distinguished values from computations and also stated that values can be used to describe computations, a question about the meaning of types arises quite naturally.
- In theory, languages like Java and Haskell often share many types, like `boolean`/`Bool`. In practice, however, these types can mean different things, if we consider the kinds of effects that the language allows.
- 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 primitive type A can be seen as $A!\{\bot, \mathsf{IO}, \mathsf{Unchecked}, \ldots\}$, because this type contains values resulting from function calls that may loop, perform IO, throw unchecked exceptions etc. The three dots signify that it can do even more, like returning a random value. We know, however, that A doesn't contain the `null` value.
- If A is not a primitive type, then it means $A!\{\bot, \mathsf{IO}, \mathsf{Unchecked}, \mathsf{Null}, \ldots\}$ - now the result can be null.
- If there's a checked exception E in the signature of a function of type $A \to B$, then its type can be interpreted as $A \to B!\{\bot, \mathsf{IO}, \mathsf{Unchecked}, \mathsf{Null}, E, \ldots\}$, 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 type a can be interpreted as $a!\{\bot, \mathsf{IO}, \mathsf{Error}, \ldots\}$, because values of this type 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).
- The type $m\ a$, where m is some monad, can be interpreted as $a!\{\bot, \mathsf{IO}, \mathsf{Error}, m, \ldots\}$, 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 type A means $A!\{\}$, because it can't have any effect - it must terminate, can't perform IO or exceptions, can't return null etc.
- The type $M\ A$ for some monad M can be interpreted as $A!\{M\}$.

An effect system

- Thanks to the distinction between values and computations and the concept of an effect, we can now state the goal of our undertaking.
- It is good to have values under control. This is the business of a type system. Types classify values, tell us how they can be constructed and used, tell us when we can compose things and prevent us from doing silly mistakes.
- Since a computation is a value and some effects, what we would like to have is therefore (loosely speaking) an effect system - a way of classifying, creating, using and composing effects.
- Before coming up with our own, let's see how such effect systems are realized in practice (we can regard any language as having an effect system, just like we can regard any language as having types, even if it's just a single one).

Where do effects come from?

- Not all languages make the same effects available.
- One could argue that, in theory, any Turing complete language can express any effect (by implementing a compiler or interpreter of a language which supports these effects). However I don't think it's true since Turing completeness is about computing pure functions, and effects are about impurity.
- Therefore, what effects are available in a language depends on the design (semantics) of that language. An effect, to be available in a language, has to have been put there, explicitly or not.

Where do effects come from: Java

- Input and output: library functions for performing IO.
- Partiality: objects of any class can be null.
- Exceptions: built-in exceptions mechanism.
- Mutable state: assignment (it's not global, but objects have internal state that can be accessed by methods).
- Randomness: implementable (impure pseudorandom generators).
- Nontermination: unrestricted while loop.

Where do effects come from: Haskell

- 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 style exceptions can be implemented using continuations.
- Mutable state: can be implemented as a monad with the type $s \to (a, s)$
- Randomness: implementable using pure pseudorandom generators. The seed can be taken from IO.
- Nontermination: unrestricted recursive calls.

Where do effects come from: Coq

- Exceptions: no Java style exceptions, but can be simulated using continuations.
- Partiality, mutable state, randomness: can be implemented like in Haskell.
- Input and output: impossible (no built-in IO).
- Nontermination: impossible because only structural recursion is allowed.

How effects are managed: Java

- 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.
- Nontermination: you can't do anything about it.

How effects are managed: Haskell

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

How effects are managed: Coq

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

A comparison of approaches

- The ad-hoc way of handling of effects, as done in Java and most mainstream languages is not too wise. It should be burnt at the stake and then forgotten.
- Monads are currently the most popular way of dealing with effects in languages which treat them seriously. The monadic techniques come in various flavours: ordinary monads, transformers, monadic classes and free monads.
- Algebraic Effects is a new idea for providing languages with an effect system. They come in two main flavours:
  - Library-level: Haskell's extensible-effects and Idris's Effect.
  - Languages-level: languages like Eff, Koka and Frank.