Effects Without Monads: Non-determinism

Back to the Meta Language

Oleg Kiselyov
Tohoku University, Japan
oleg@okmij.org

Abstract

Surprisingly, we could write interesting non-deterministic programs in an ML-like language just as naturally and elegantly as in the functional-logic language Curry – ML's call-by-value and the lack of support for monads notwithstanding. The key idea goes back to the very origins of ML: write non-deterministic computations in a small, tagless-final embedded DSL, with ML playing the role of a 'preprocessor'. What is new and unexpected is how well our experiment turned out: we even got by with a first-order, simple to reason and implement DSL – compensated by the richness of the Meta Language. Forsaking monads permitted more DSL implementations. One wonders how many more practically interesting problems can be solved with such embarrassingly simple DSLs.

1. Summary

No talk about effects nowadays can avoid monads. Monads have been introduced to ML more [8] or less [1] formally and underlie the widely used OCaml libraries Lwt and Async. Yet effects are not married to monads and approachable directly. The structuring, the separation of 'pure' (context-independent) and effectful computations [5] can be done without explicating mathematical monads, and especially without resorting to vernacular monads such as State, etc. We present an example: a simple, effectful, domain-specific sublanguage embedded into an expressive 'macro' metalanguage. Abstraction facilities of the metalanguage such higher-order functions and modules help keep the DSL to the bare minimum, often to the first order, easier to reason about and implement.

The key insight predates monads [9] and goes all the way back to the origins of ML, as a scripting language for the Edinburgh LCF theorem prover [3]. What has not been clear is how simple an effectful DSL may be while remaining useful. How convenient it is, especially compared to the monadic encodings. How viable it is to forsake the generality of first-class functions and monads and what benefits it may bring. We report on an experiment set out to explore these questions.

We pick a rather complex effect – non-determinism – and use it in OCaml, which may seem unsuitable since it is call-by-value and has no monadic sugar. And yet, we can write non-deterministic programs just as naturally and elegantly as in Haskell or Curry.

The running tutorial example is computing all permutations of a given list of integers. Albeit a simple exercise, the code is often rather messy and not *obviously* correct. In the functional-logic language Curry, it is strikingly elegant: mere `foldr insert []`. It is the re-statement of the specification: a permutation is moving the elements of the source list one-by-one into *some* position in the initially empty list. From its very conception in 1959 [7], non-determinism was called for to write clear specifications – and then to make them executable. That is what will shall do.

This extended abstract summarizes the complete development and code at `http://okmij.org/ftp/tagless-final/nondet-effect.html`.

2. Non-determinism through a DSL

We start by designing a language just expressive enough for our problem of computing a list permutation using non-determinism. We embed this "domain-specific" language (DSL) into OCaml in the tagless-final style. (Instead of OCaml, we could have used any other ML or ML-like language.) In the tagless-final style [4], a DSL is defined by specifying how to compute the meaning of its expressions. The meaning is an OCaml value of some abstract type (such as the types `int_t` and `ilist_t` below, the semantic domains of integer and integer list expressions). The meaning of a complex expression is computed by combining the meaning of immediate sub-expressions, that is, compositionally. A language is thus defined by specifying the semantic domain types and the meaning computations for its syntactic forms. These definitions are typically collected into a signature, such as:

```
module type NDet = sig
```

which we will be filling in. Since we will be talking about integer lists, we need the integer type and at least the integer literals:

```
  type int_t
  val int : int → int_t
```

We can add the standard operations on integers, but they are not needed for the problem at hand. They can always be added later. After all, the extensibility is the strong suit of the tagless-final embedding.

We also need integer lists, with the familiar constructors:

```
  type ilist_t
  val nil :    ilist_t
  val cons: int_t → ilist_t → ilist_t
  val list : int list → ilist_t
```

The `list` primitive makes an OCaml list to be the list in our DSL: although every DSL list can be expressed through `nil` and `cons`, the special notation for literal DSL lists is convenient. We also need pattern-matching on lists, or the deconstructor. The syntax is admittedly ungainly: we are trying to represent **match** ... **with** as an applicative expression:

```
  val decon: ilist_t →
    ( unit → ilist_t ) →                (* if nil *)
    ( int_t → ilist_t → ilist_t ) → (* if cons h t *)
     ilist_t
```

We also need `foldr`. Strictly speaking, we do not need it: it is expressible through the features already defined. But `foldr` is so fundamental, it is convenient to have as a primitive.

```
  val foldr : ( int_t → ilist_t → ilist_t )→ ilist_t → ilist_t → ilist_t
```

Finally, we define the operations for non-determinism: failure and the binary choice. The latter non-deterministically executes one of its arguments.

```
val fail : ilist_t
val (|||): ilist_t → ilist_t → ilist_t
```

And we are done.

An attentive reader may get the feeling that something is amiss: where are functions? We have defined neither the DSL function type nor operations to create and apply functions. Our DSL is not a lambda-calculus; it is essentially first order. Please hold your wonder.

3. List permutation, Non-deterministically

However feeble our NDet DSL may be, it is enough for the task at hand. We now use it to write the list permutation as elegantly as in Curry.

First, we need the non-deterministic list insertion: insert x lst is to insert the element x *somewhere* in lst, returning the extended list. That is, it inserts x at the front of lst, or after the first element of lst, or after the second element of lst, etc. The algorithm can be formulated, and hence implemented, inductively: insert x lst either inserts x at the front of lst or within lst, i.e., somewhere in its tail. Computing the list permutation is now accomplished. The following is the complete code, which also includes a simple test.

```
module Perm(S:NDet) = struct open S
  let rec insert x l =
    cons x l ||| decon l
                 (fun () → fail )
                 (fun h t → cons h ( insert x t))
  let perm = foldr insert nil
  let test1 = perm (list [1;2;3])
end
```

The DSL primitives such as foldr, fail, nil etc. are all defined in the implementation S of the signature NDet. The code does not depend on any particular implementation, which is hence abstracted over as an argument S. DSL code is hence typically represented as an OCaml functor, parameterized by the DSL implementation.

Although our code looks like the Curry code and is exceedingly simple, there is something odd about it. We have said that NDet has no functions: no function types, no way to create or apply functions, let alone recursive functions. What is insert then? Isn't foldr a higher-order function? They are functions – in the *metalanguage* but not in NDet. From the DSL point of view, insert is a 'macro'. Our code then is a combination of a trivial, non-deterministic DSL with a very expressive, higher-order 'macro' system[1]. Moreover, the DSL evaluation and the 'macro-expansion' run like coroutines. It is not unheard of: after all, coroutines were invented as a communication mechanism among phases of a Cobol compiler [2]. This coroutine-like evaluation is the essence of Moggi's computational calculus [5].

One often hears (from the reviewers) the complaint that writing DSL expressions as functors is cumbersome. But there are other ways, blending the DSL code into the regular OCaml. The result looks quite like the Lightweight Modular Staging (LMS) in Scala, which has been used for serious DSLs[2]:

```
let perm : (module NDetO) → int list → int list list =
fun (module S:NDetO) l → let open S in
  let rec insert x l =
    cons x l ||| decon l
                 (fun () → fail )
                 (fun h t → cons h ( insert x t))
  in run @@ foldr insert nil ( list l)
```

Modular implicits, currently an OCaml branch, save us the trouble of passing the NDet implementation explicitly. DSLs become convenient: DSL primitives look like the ordinary OCaml operations, but can be distinguished by their types. Instead of first-class modules we could have used plain records. Our approach hence easily applies to other ML(-like) languages.

[1] An old joke comes to mind: "Much of the power of C comes from having a powerful preprocessor. The preprocessor is called a programmer." [6].

[2] Here, NDet0 is NDet extended with the observation function run. See the accompanying code for details.

4. Implementations of Non-determinism

To run the Perm code we need an implementation of the NDet signature. Since we are interested in the list of all permutations, it is natural to take int_t and ilist_t to be lists of all choices an integer or a list DSL expression may produce.

```
type int_t = int list
type ilist_t = int list list
```

(See the accompanying code for the full implementation).

This is the List monad! Yes, it is – after all, it is one of the implementations of non-determinism, envisioned already by Rabin and Scott in the the 1950s. *It is not the only one* (see §5). The accompanying code shows another implementation of NDet, in terms of delimited continuations, the delimcc library. One may easily think of others, e.g., using the operating system threads.

5. When Monads will not do

Our NDet DSL is not monadic. We program in it just like in ML, directly operating on effectful expressions, without any binds and returns. To be sure, monads are not without benefits: clear separation of pure and effectful computations in types and syntax; uniformity; easy extension to higher-order functions. Let us see how much if any we have lost without monads.

Our DSL approach just as clearly separates pure and effectful: anything of the type int_t or ilist_t is potentially non-deterministic; everything else is deterministic. Thus from the type of insert : int_t → ilist_t → ilist_t we immediately tell that insert deterministically combines non-deterministic computations.

Thanks to the richness of the metalanguage, NDet did not need higher-order (or even first-order) functions. The experience with LMS shows that for rather many practical problems, a first-order DSL is sufficient.

Finally, forsaking monads permits more implementations of our NDet. Suppose we wish to generate code for list permutations and realize the types as

```
type int_t = int list code
type ilist_t = int list list code
```

(see the accompanying code for the full implementation). Had NDet been monadic, such an implementation is impossible: α code is not a monad. First, return : $\alpha \to \alpha$ code is problematic since not every value can be lifted to code (think of reference cells and I/O channels). Second, we cannot generally execute the code until we have finished generating it (because intermediate code may be open). Therefore, bind is not expressible either.

6. Conclusions

All in all, we have described a direct alternative to the monadic encoding of effects: defining a small domain-specific language with the necessary effectful operations. The DSL will be blended into OCaml or other ML-like language; therefore, it can be kept tiny, with no abstraction facilities of its own, or even functions. OCaml, serving as an inordinarily expressive macro language, compensates.

We have reported only one experiment, which – combined with the related LMS experience – suggests that the direct encoding of effects is viable. More experiments are needed to better grasp its usefulness. Specifically we would like to try examples in the scope of Async or Lwt libraries.

A. Advanced non-determinism: Sorting

An immediate application of list permutation is sorting: sorting, by definition, is a sorted permutation. Our DSL can truly specify sort just like that – and execute it, too. It is called 'slow sort' – one of the benchmarks of functional-logic programming. Although not usually fast, it is correct by definition. The actual performance depends on the implementation and could be quite good.

To express sorting we need two more non-deterministic primitives. Extending a language defined in the tagless-final style is easy, by adding new definitions and reusing the old ones:

```
module type NDetComm = sig
  include NDet
  val rld  : (int list → bool) → ilist_t → ilist_t
  val once : ilist_t → ilist_t
end
```

The operation rld is a form of a logical conditional: it imposes a guard (a predicate constraint) on a non-deterministic expression. It is hence akin to List.filter. The name is chosen to match the Curry standard library. The primitive once (called head in Curry) expresses the so-called 'don't care non-determinism': if an expression has several latent choices, once picks one of them.

The sorting is written literally as "*a* sorted permutation":

```
module Sort(Nd:NDetComm) = struct
  open Nd
  include Perm(Nd)
  let rec sorted = function
    | []  → true
    | [_] → true
    | h1 :: h2 :: t → h1 ≥ h2 && sorted (h2::t)

  let sort l = once @@ rld sorted @@ perm l
  let tests = sort ( list [3;1;4;1;5;9;2])
```

An attentive reader must have noticed that the sortedness is expressed 'meta-theoretically' as one might say (why?).

Extending a DSL implementation is just as easy as extending the language definition: we just add the code for the new primitives, which are indeed primitive:

```
module NDetLComm = struct
  include NDetL
  let rld  = List. filter
  let once = function [] → [] | h :: _ → [h]
```

We can really sort: `let module M = Sort(NDetLComm) in M.tests`

B. Exercises

An interested reader might want to ponder:

1. We have said that foldr is not actually necessary: it can be written using the other features of Ndet. Write it.
2. Typically, a tagless-final presentation features the type α repr, a set of OCaml values that represent DSL expressions of the type α. We have managed to do without α repr. What have we lost?
3. Generalize the NDet signature introducing α repr and implement this language.
4. Does it make sense to define a separate type for values and expressions of our DSL? What benefits it may confer?
5. Add yet another implementation of NDet: e.g., using the free(r) monad or threads. Besides the depth-first search (underlying the list implementation), try to implement complete search strategies such as breadth-first search or iterative deepening.
6. Implement other classical non-deterministic puzzles from the Curry example library `http://www.informatik.uni-kiel.de/~mh/curry/examples/`
7. The slow sort is particularly slow in the shown list implementation of NDet. Why? How to speed it up?

References

[1] J. Carette and O. Kiselyov. Multi-stage programming with functors and monads: Eliminating abstraction overhead from generic code. *Science of Computer Programming*, 76(5):349–375, 2011.

[2] M. E. Conway. Design of a separable transition-diagram compiler. *Commun. ACM*, 6(7):396–408, 1963. .

[3] M. Gordon, R. Milner, L. Morris, M. Newey, and C. Wadsworth. A metalanguage for interactive proof in LCF. In *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages*, pages 119–130, Tucson, Arizona, Jan. 23–25, 1978. ACM SIGACT-SIGPLAN. URL `\url{http://www-public.tem-tsp.eu/~gibson/Teaching/CSC4504/ReadingMaterial/GordonMMNW78.pdf}`.

[4] O. Kiselyov. Typed tagless final interpreters. In *Proceedings of the 2010 International Spring School Conference on Generic and Indexed Programming*, SSGIP'10, pages 130–174. Springer-Verlag, Berlin, Heidelberg, 2012. ISBN 978-3-642-32201-3. . URL `http://dx.doi.org/10.1007/978-3-642-32202-0_3`.

[5] E. Moggi and S. Fagorzi. A monadic multi-stage metalanguage. In A. D. Gordon, editor, *Proceedings of FoSSaCS 2003: Foundations of Software Science and Computational Structures, 6th International Conference*, number 2620 in LNCS, pages 358–374. Springer, 7–11 Apr. 2003. ISBN 3-540-00897-7. URL `http://www.disi.unige.it/person/MoggiE/ftp/fossacs03.pdf`.

[6] P. J. Moylan. The case against C. Technical Report TR–EE9240, Centre for Industrial Control Science, Department of Electrical and Computer Engineering, University of Newcastle, Australia, 1992.

[7] M. O. Rabin and D. Scott. Finite automata and their decision problems. *IBM Journal of Research and Development*, 3:114–125, 1959.

[8] N. Swamy, N. Guts, D. Leijen, and M. Hicks. Lightweight monadic programming in ML. In *ICFP'11*, pages 15–27, Sept. 2011.

[9] M. Wand. Specifications, models, and implementations of data abstractions. *Theoretical Computer Science*, 20(1):3–32, Mar. 1982.