Effects Without Monads: Non-determinism Back to the Meta Language

Oleg Kiselyov
Tohoku University, Japan
oleg@okmij.org

We reflect on programming with complicated effects, recalling an undeservingly forgotten alternative to monadic programming and checking to see how well it can actually work in modern functional languages.

We adopt and argue the position of factoring an effectful program into a first-order effectful DSL with a rich, higher-order 'macro' system. Not all programs can be thus factored. Although the approach is not general-purpose, it does admit interesting programs. The effectful DSL is likewise rather problem-specific and lacks general-purpose monadic composition, or even functions. On the upside, it expresses the problem elegantly, is simple to implement and reason about, and lends itself to non-standard interpretations such as code generation (compilation) and abstract interpretation. A specialized DSL is liable to be frequently extended; the experience with the tagless-final style of DSL embedding shown that the DSL evolution can be made painless, with the maximum code reuse.

We illustrate the argument on a simple but representative example of a rather complicated effect – non-determinism, including committed choice. Unexpectedly, it turns out we can write interesting non-deterministic programs in an ML-like language just as naturally and elegantly as in the functional-logic language Curry – and not only run them but also statically analyze, optimize and compile. The richness of the Meta Language does, in reality, compensate for the simplicity of the effectful DSL.

The key idea goes back to the origins of ML as the Meta Language for the Edinburgh LCF theorem prover. Instead of using ML to build theorems, we now build (DSL) programs.

1 Introduction

How to cope with the complexity of writing programs? How to structure computations? Many methodologies have been proposed over the decades: procedures, structured programming, OOP, AOP, algebraic specifications and modules, higher-order functions, laziness – and, lately, monads and their many generalizations. Although monads are not the only way to organize (effectful) computations, they are by all accounts receiving disproportionate attention (just do a quick Google search). In ML, monads have been introduced more [38] or less [7] formally and underlie the widely used OCaml libraries Lwt[1] and Async.[2]

This position paper seeks to draw attention to a non-monadic alternative: Rather than structuring (effectful) programs as monads – or applicatives, arrows, etc., – we approach programming as a (micro) language design. We determine what data structures and (effectful) operations seem indispensable for the problem at hand – and design a no-frills language with just these domain-specific features. We embed this bare-bone DSL into OCaml, relying on OCaml's extensive facilities for abstraction and program composition (modules, objects, higher-order functions), as well as on its parsing and type checking.

[1] http://ocsigen.org/lwt/
[2] https://github.com/janestreet/async

Sam Lindley and Gabriel Scherer (Eds.):
ML Family / OCaml Users and Developers workshops 2017
EPTCS 294, 2019, pp. 15–40, doi:10.4204/EPTCS.294.2

We state the main points of our argument in §1.2. We hasten to say that the key insight is rather old [41], quite resembling algebraic specifications (see Wirsing's comprehensive survey [43]). In fact, it is the insight behind the original ML, as a scripting language for the Edinburgh LCF theorem prover [14] – only applied to programs rather than theorems. What has not been clear is how simple an effectful DSL may be while remaining useful. How convenient is it, especially compared to the monadic encodings? How viable is it to forsake the generality of first-class functions and monads and what benefits may come? We report on an experiment set out to explore these questions.

The present paper follows the style of Hughes [20], Hudak [18] and Goguen [13] – or, for mathematically inclined, Pólya [32] – arguing from examples. Just like those arguments, it is indeed hard to grasp the limitations and applicability, and it is hard to formalize. Problem solving in general is a skill to learn rather than an algorithm to implement; it is inherently informal. Even in mathematics, how to prove theorems is an art and a judgement; one acquires it not by following rigorous descriptions but by reading existing proofs and doing exercises. This is the format we follow in the paper.

1.1 Motivation

This present paper comes as the result of decade-long long experience with the tagless-final style of DSL embedding [24] and the re-discovering[3] and polishing of extensible effects ([26, 25]). It was prompted however by the following message, posted on the Caml-list by Christoph Höger in March 2017:[4]

> "Assume a simple OCaml program with two primitives that can cause side-effects:
>
> ```
> let counter = ref 0
> let incr x = counter := !counter + x ; !counter
> let put n = counter := n; !counter
> put (5 + let f x = incr x in f 3)
> ```
>
> This example can be transformed into a pure program using a counter monad (using ppx_monadic syntax):
>
> ```
> do_;
>  i ⟵ let f x = incr x in f 3 ;
>  p ⟵ put (5 + i)
>  return p
> ```
>
> For a suitable definition of bind and return, both programs behave equivalently. My question is: How can one automatically translate a program of the former kind to the latter?"

The message left me puzzled about its author's goals and motivations. It is hard to imagine he preferred the monadic program for its verbose notation, replete with irrelevant names like i and p. Was the author after purity? That is a mirage we lay bare in §6. Was he attracted to the separation of effectful and non-effectful code and the possibility of multiple interpretations of effects ('the overriding of the semicolon')? These good properties are not unique to monads. The other, often forgotten, ways of dealing with side-effects ought to be more widely known.

The second cue came about a month later, observing students solving an exercise to compute all permutations of a given list of integers. The reader may want to try doing that in their favorite language. Albeit a simple exercise, the code is often rather messy and not *obviously* correct. In the functional-logic

[3] `http://okmij.org/ftp/Computation/having-effect.html`

[4] Christoph Höger: Transforming side-effects to a monad. Posted to `caml-list@inria.fr` on Thu, 23 Mar 2017 20:56:16 +0100

language Curry [15] built around non-determinism, computing a permutation 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. The code immediately tells that the number of possible permutations (possible choices of permutations) of n elements is n!. From its very conception in the 1950s [33], non-determinism was called for to write clear specifications – and then to make them executable. Can we write the list permutation code just as elegantly in a language that was not designed with non-determinism in mind and offers no support for it? How far can we extend it?

1.2 Points to Argue

The primary goal for the paper is to report on the experiment set out to explore the viability and consequences of a particular method of writing effectful programs. Although all the ingredients have been known (some of them for so long that they are almost forgotten), how well the approach actually works for interesting problems can only be determined empirically.

Along with the describing the experiment and its ramifications, we also offer an argument: why this approach is worth exploring in the first place. The points of the argument, reverberating throughout the paper, are collected below:

Effects are not married to monads The discussion after the first presentation of this paper, at the ML Family workshop 2017, was one of many indications that monads have a special, almost cult status in the minds of functional programmers. There is no doubt that monads clearly delineate effectful computations, in syntax and in types, and offer the reasoning principles (equational laws) about effectful programs. What many do not seem to realize is that these benefits are not unique to monads, or that not all effects are expressible with monads,[5] or that the flexibility of the monadic encoding ('overriding semicolon') is limited. Code generation and abstract interpretation, for example, do not fit the monadic framework (see §5).

Although this point may be obvious to some, the caml-list (as noted in §1.1), Reddit, Stack Overflow, etc. discussion places are awash with misunderstandings and irrational exuberance towards monads.[6] The argument pointing out their proper place and limitations is worth repeating.

Separate rather than combine higher-order and effects The present paper is an exploration of a less common approach to writing reusable, properly abstracted effectful programs. Rather than combining effectful operations with modules, objects, higher-order functions, we separate them. First we determine the data types and operations needed for the problem at hand, and define the corresponding domain-specific language (DSL). The language is often first-order, and its operations have a number of *domain-specific* effects (such as references to application-specific context, communication, logging, etc).

Since the DSL is intentionally without abstraction or syntactic sugar – just enough to express the problem at hand, however ungainly – programming in it directly is a chore. That is why we endow it with a very expressive 'preprocessor', by embedding into a metalanguage with rich abstractions like functions, definitions, modules, etc.

[5] The fact that not all effects are expressible as monads was noted already by Wadler [40]. That fact has motivated the development of various monad-*like* (relative monads [2], parameterized monads [3]) and unlike interfaces (applicatives [27] and arrows [19]).

[6] Here is a small sample of links `https://stackoverflow.com/questions/44965/what-is-a-monad` `https://news.ycombinator.com/item?id=16422452` `https://news.ycombinator.com/item?id=17645277` `https://news.ycombinator.com/item?id=16419877`

Not all problems can be thus factored into a first-order DSL and a higher-order metalanguage (e.g., the factoring does not support arbitrary higher-order effectful functions). Therefore, how well the factoring works in practice and if it is worth paying attention to become empirical questions. The paper describes one case study, exploring how far we can push this approach.

General vs. specific: it is a trade-off The approach to be evaluated in the present paper – the factoring of an effectful program into a simple DSL and a rich preprocessor – is not general purpose. The non-deterministic DSL is not general purpose either: we introduce only those data types and operations that are needed for the problem at hand and its close variations. Domain-specific, narrow solutions should not be always looked down upon, we argue.

The compelling case for (embedded) DSLs has been already made, by Hudak [18]. The present paper is another case study. We also demonstrate, in §5, one more advantage of an embedded DSL: the ability to evaluate the same DSL code in several ways. We can not only (slowly) interpret the code, but also perform static analyses such as abstract interpretation, and generate (faster) code.

The advantages of the domain-specific approach have to be balanced against the applicability (how wide is the domain, does it let us do something interesting or practically significant) and extensibility (how easy it is to extend the program and the domain and reuse the existing code). This is a trade-off, which the case study in the present paper is to help evaluate.

Thus we do not argue that the domain-specific approach is 'better'. We do argue, however, against the presumption (evidenced in the received comments on the drafts of this paper) that one always has to strive for the general solution. Premature generalization and abstraction, like premature optimization, is not a virtue.

Try domain-specific first When deciding which approach to effects fits the problem at hand, we advocate trying a specialized solution (such as the DSL factoring) first. Typically, whether it works out or not becomes clear very soon; if it does not, little effort is wasted because the approach is so simple.

The structure of the paper is as follows. §2 describes the main experiment: can we write the list permutation in OCaml as elegantly as in Curry. Specifically, §2.1 introduces the intentionally very simple, essentially first-order, DSL for the specific domain of non-deterministic computations on integer lists, and §2.2 uses it to express the list permutation. The readers can see for themselves how good (simple, understandable, close to Curry) it looks. A way to make the DSL embedding seamless is described in §2.3. Several 'standard' implementations of the DSL are discussed in §3, whereas §4 extends the DSL and its implementation with the committed choice and presents another classical Curry example: slow sort. Thus the factoring of an effectful computation into a first-order DSL and a powerful metalanguage turns out to be viable – an outcome that was not at all clear at the beginning.

Our language is truly domain-specific: for example, it offers no abstraction mechanism of its own and no general monadic interface for writing effectful computations. As an upside, the DSL admits useful non-standard implementations. §5 shows three. In particular, §5.2 describes an abstract interpretation, to statically estimate the degree of non-determinism of a DSL term. The code-generation interpretation – the DSL compiler – is presented in §5.3.

§6 discusses the presented factoring approach, answering several commonly heard objections. The main theoretical ideas have all been known in isolation, often for many decades, as we review in §7.

The source code of all our examples is available at `http://okmij.org/ftp/tagless-final/nondet/`.

2 Non-determinism through a DSL

This section introduces the DSL that lets us program all permutations of a given list of integers in the same starkly elegant way it is done in Curry:

```
perm = foldr insert []
```

This running example, although rather simple (and hence easy to explain and thoroughly examine), distills large, practical projects such as machine-learning [36] or the conduct of clinical trials [29]. The example is also interesting because it deals with a rather complex effect – non-determinism, which is rarely supported natively. Yet we are able to use non-determinism just as easily as in the language Curry, specifically designed for non-determinism. We manage with only OCaml at our disposal, which may seem unsuitable since it is call-by-value and has no monadic sugar.

§2.1 defines the DSL, §2.2 writes our example in it, and §2.3 polishes the DSL embedding by overcoming the ungainly (and objectionable, to some) functors. The (standard) implementations of the DSL are discussed in §3 – and three non-standard ones in §5.

2.1 DSL Definition

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 – or Scala or Haskell or Rust.) Recall, in the tagless-final style a DSL is defined by specifying how to compute the meaning of its expressions [24]. The meaning is represented by 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 meanings of its immediate subexpressions, 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
  type int_t
  val int: int → int_t

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

  val recur:     (* recur c n lst: see text for the explanation *)
    (int_t * ilist_t → (unit → ilist_t) → ilist_t) → ilist_t → ilist_t →
    ilist_t

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

Since we will be talking about integer lists, we need the integer type int_t and at least the integer literals. Whereas 1 is an OCaml integer literal, the OCaml expression int 1 represents a DSL integer literal. We do not define any operations on integers, since they are not needed for the problem at hand. They can

always be added later. After all, the ease of extending the language with new types and operations is the strong suit of the tagless-final embedding.

We also need integer lists, with the familiar constructors nil and cons. The list operation turns an OCaml list into a list in our DSL: list [1;2;3] (compare with int 1 example above). Although every DSL list can be expressed through nil and cons, the special notation for literal DSL lists is convenient.

We also need a way to recursively analyze/deconstruct lists. For that purpose, we introduce the recursor recur, inspired by the recursor R for natural numbers in Gödel's System T (see Tait [39] for the modern exposition; Tait calls R an iteration). Similarly to R, the meaning of our recur is specified by the following equalities (or, algebraic identities):

$$\begin{aligned} \mathsf{recur}\ c\ n\ \mathsf{nil} &\equiv n \\ \mathsf{recur}\ c\ n\ (\mathsf{cons}\ h\ t) &\equiv c\ (h,t)\ (\mathbf{fun}\ ()\ \to\ \mathsf{recur}\ c\ n\ t) \end{aligned} \tag{1}$$

As in high-school algebra, an identity states that the two terms connected by the $\equiv$ sign are to be considered 'the same'. If an identity contains variables (such as c, n, etc. above – typeset in the mathematical font), it should hold for all instantiations, i.e., replacements of a variable with a term of a suitable type. One may bet that the thunk **fun** $() \to \ldots$ visible in (1) was not present in Gödel's formulation of R. Why we have introduced it in our recur will become clear in the next section.

Finally, the NDet signature defines the operations for non-determinism: failure and the binary choice. Specifically, l1 ||| l2 denotes a non-deterministic choice among two lists, l1 and l2. To make sure the operations l1 ||| l2 and fail, however they may end up being implemented, agree with the intuitions about the non-deterministic choice and failure, we impose the following identities:

$$\begin{aligned} \mathsf{cons}\ x\ \mathsf{fail} &\equiv \mathsf{fail} \\ \mathsf{cons}\ x\ (l_1\ |||\ l_2) &\equiv \mathsf{cons}\ x\ l_1\ |||\ \mathsf{cons}\ x\ l_2 \\ \mathsf{recur}\ c\ n\ \mathsf{fail} &\equiv \mathsf{fail} \\ \mathsf{recur}\ c\ n\ (l_1\ |||\ l_2) &\equiv \mathsf{recur}\ c\ n\ l_1\ |||\ \mathsf{recur}\ c\ n\ l_2 \\ (x\ |||\ y)\ |||\ z &\equiv x\ |||\ (y\ |||\ z) \end{aligned} \tag{2}$$

Any implementation of NDet is supposed to verify that the above identities hold for that implementation. In OCaml, we cannot check the satisfaction mechanically; we cannot even attach these identities to the signature except in comments. Wirsing's survey [43] cites many systems which do verify the satisfaction of equational specifications.

An attentive reader may get the feeling that something is amiss: the NDet DSL does not look at all like a functional language. There are no function types (only integers and integer lists) and hence no operations to construct, or even apply, functions. NDet is *not* a lambda-calculus. How useful can such a trivial language be? On the other hand, isn't recur a higher-order function, from the look of the type of its first argument? Please hold your wonder.

Exercise 1 *When one hears about recursively deconstructing a list, what is likely to spring to mind is foldr. Yet for some reason we introduced the relatively obscure recur instead. Can you venture a guess why we did that? How does recur relate to foldr?*

Exercise 2 *Does it make sense to define separate types for values and expressions of our DSL? What benefits may come from this separation?*

Exercise 3 *The signature NDet is not algebraic (why?). How would you characterize it?*

Exercise 4 *The identities* (2) *are by no means the complete equational specification of non-determinism. What other identities with fail and ||| could be added to* (2)*?*

2.2 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. For reference, here is the permutation code as it appears in the Curry standard library:

```
insert x [] = [x]
insert x (y:ys) = (x:y:ys) ? (y:insert x ys)

perm = foldr insert []
```

To realize this code in the NDet DSL, we first tackle 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 written in the NDet DSL, which also includes a simple test.[7]

```
module Perm(S:NDet) = struct
  open S

  (* val foldr: (int_t → ilist_t → ilist_t) → ilist_t → ilist_t → ilist_t *)
  let foldr c = recur (fun (h,_) r → c h (r ()))

  let insert x =
   recur (fun (h,t) inserted → cons x (cons h t) ||| cons h (inserted ()))
         (cons x nil)

 let perm = foldr insert nil
 let test1 = perm (list [1;2;3])
end
```

The DSL primitives such as recur, cons, 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. Therefore, the DSL code is typically represented as an OCaml functor, parameterized by the DSL implementation (there are nicer-looking representations, please wait till §2.3). Since NDet only provides recur but no foldr, first we have to implement the latter (with the expected inferred type shown in the comment). The implementation is straightforward. The insert is straightforward as well, mirroring the Curry code (keeping in mind that the nondeterministic-choice operator is spelled ? in Curry and ||| in our code). The code keeps the invariant that inserted (), denoting the recursive invocation of insert, is the expression returning the list with exactly one x inserted somewhere. The same invariant is true of the Curry code.

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. What is insert then? Isn't foldr a higher-order function? They are functions – in the *metalanguage*, rather than in NDet. We use the higher-order facilities of OCaml to construct first-order DSL expressions. OCaml truly acts as a preprocessor for the DSL; insert is hence a 'macro'. Our code then is a combination of

[7] The accompanying code includes many more (regression) tests.

a trivial, non-deterministic DSL with a very expressive, higher-order 'macro' system.[8] 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 [9]. The coroutining between a lambda-calculus–based 'metalanguage' and the embedded 'effectful' language is the essence of Reynolds' Idealized Algol [34] and Moggi's computational calculus [28].

To get a better feeling for the "macro-expansion" and also the confidence in the DSL, it is worth doing a simple exercise: determine the DSL terms that should be identical to perm (list [1;2;3]). Below we do a part of the exercise, working out the identities of insert (int 1) (list [2;3]). For the sake of readability, we write DSL terms like int 1 as $\overline{1}$ and DSL list literals like cons (int 2) (cons (int 3) nil) as $\overline{[2;3]}$.

insert $\overline{1}$ $\overline{[2;3]}$
$\equiv$ *(∗ inlining definitions: "macro−expansion" ∗)*
recur (**fun** (h,t) inserted $\rightarrow$ cons $\overline{1}$ (cons h t) ||| cons h (inserted ())) $\overline{[1]}$ $\overline{[2;3]}$
$\equiv$ *(∗ identities* (1) *∗)*
(**fun** (h,t) inserted $\rightarrow$ cons $\overline{1}$ (cons h t) ||| cons h (inserted ())) ($\overline{2}$,$\overline{[3]}$)
  (**fun** () $\rightarrow$ recur (**fun** (h,t) inserted $\rightarrow$ ...) $\overline{[1]}$ $\overline{[3]}$)
$\equiv$ *(∗ substitution of values ∗)*
cons $\overline{1}$ (cons $\overline{2}$ $\overline{[3]}$) ||| cons $\overline{2}$ (recur (**fun** (h,t) inserted $\rightarrow$ ...) $\overline{[1]}$ $\overline{[3]}$)
$\equiv$ *(∗ convention for the literal lists ∗)*
$\overline{[1;2;3]}$ ||| cons $\overline{2}$ (recur (**fun** (h,t) inserted $\rightarrow$ ...) $\overline{[1]}$ $\overline{[3]}$)
$\equiv$ *(∗ once again identities* (1) *∗)*
$\overline{[1;2;3]}$ ||| cons $\overline{2}$ (cons $\overline{1}$ (cons $\overline{3}$ $\overline{[]}$) ||| cons $\overline{3}$ (recur (**fun** (h,t) inserted $\rightarrow$ ...) $\overline{[1]}$ $\overline{[]}$))
$\equiv$ *(∗ and again ∗)*
$\overline{[1;2;3]}$ ||| cons $\overline{2}$ ($\overline{[1;3]}$ ||| $\overline{[3;1]}$)
$\equiv$ *(∗ identities* (2) *∗)*
$\overline{[1;2;3]}$ ||| ($\overline{[2;1;3]}$ ||| $\overline{[2;3;1]}$)

The identities ought to hold in any implementation of NDet. Thus, whatever the implementation, insert $\overline{1}$ $\overline{[2;3]}$ should amount to the choice among $\overline{[1;2;3]}$, $\overline{[2;1;3]}$ and $\overline{[2;3;1]}$, in full agreement with our intuitions.

We have used $\equiv$ to mean the least equivalence relation that contains the identities (1) and (2), and is closed under substitutions of OCaml values into OCaml lambda-terms. In other words, $\equiv$ includes the "macro-expansion" performed as the ordinary OCaml call-by-value evaluation. The thunk **fun** () $\rightarrow$ recur *c n t* in (1) was needed precisely for the sake of this value substitution.

Exercise 5 *Complete the exercise and work out perm (list [1;2;3]).*

2.3 Smoother DSL Embedding

One often hears 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 [35] – the metaprogramming, DSL-embedding framework which has been used for 'industrial-strength' DSLs.

As a warm-up, let us take one particular DSL implementation, such as NDetL to be described in §3. Let us write perm without any functors this time, as an ordinary OCaml function:

[8] An old joke comes to mind: "Much of the power of C comes from having a powerful preprocessor. The preprocessor is called a programmer." [30].

```
let perm : int list → int list list = fun l →
  let open NDetL in
  let foldr c = recur (fun (h,_) r → c h (r ())) in
  let insert x =
   recur (fun (h,t) r → cons x (cons h t) ||| cons h (r ())) (cons x nil)
  in foldr insert nil (list l)
```

This perm is truly an ordinary OCaml function, to be applied as perm [1;2;3].

We now abstract over the DSL implementation. First, we add to NDet the observation operation, so we may *generically* extract the the list of permutation choices from the result of the perm computation. (One may argue that such a run operation should have been a part of NDet. On the other hand, we shall demonstrate non-standard interpretations of NDet, whose results are not permutation lists but rather static analyses of the generated code.)

```
module type NDetO = sig
  include NDet
  val run : ilist_t → int list list
end
```

The permutation function will receive the DSL implementation as the (first-class) module argument:[9]

```
let perm : (module NDetO) → int list → int list list = fun (module S:NDetO) l →
  let open S in
  let foldr c = recur (fun (h,_) r → c h (r ())) in
  let insert x =
   recur (fun (h,t) r → cons x (cons h t) ||| cons h (r ())) (cons x nil)
  in run @@ foldr insert nil (list l)
```

Modular implicits [42] can even 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 therefore easily applies to other ML(-like) languages.

3 Implementing Non-determinism

To run the Perm code we need an implementation of the NDet signature. There are many of them, even in this paper (see the exercises at the end of the section, and §5). We start with the 'list of successes', the most familiar model of non-determinism, envisioned already by Rabin and Scott in the 1950s [33]. This model is also called 'list monad'; our code, however, does not use the monad in its full generality, as we shall see soon. The realizations of NDet in §5 cannot be expressed as monads at all, to be explained there.

In this list implementation, to be called NDetL, ilist_t is the list of all choices that a list DSL expression may produce:

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

[9] The right-associative infix operator @@ of low precedence is application: f @@ x + 1 is the same as f (x + 1) but avoids the parentheses. The operator is the analogue of $ in Haskell.

We are talking about OCaml lists, which are finite and 'eager'. Generally, this is not the best choice for performance; however, this realization fits very well our running example, which is to compute the list of all possible permutation choices. (We shall encounter this interplay of generality and specialization many more times.) Again, we are interested in non-deterministic computations on integer lists only; DSL integers are always deterministic and therefore, can be represented as plain OCaml int. All in all, the NDetL implementation is as follows:

```
module NDetL = struct
  type int_t = int
  let int x = x

  let concatmap: (α → β list) → α list → β list =
    fun f l → List.concat @@ List.map f l

  type ilist_t = int list list
  let nil = [[]]
  let cons: int_t → ilist_t → ilist_t =
    fun x → List.map (fun l → x::l)
  let list x = [x]

  let rec recur: (int_t * ilist_t → (unit → ilist_t) → ilist_t) → ilist_t → ilist_t → ilist_t =
    fun f z → concatmap @@ function
      | []   → z
      | h::t → f (int h, list t) (fun () → recur f z (list t))

  let fail: ilist_t = []
  let (|||): ilist_t → ilist_t → ilist_t = (@)
end
```

As expected, literal list expressions such as list and nil are deterministic: have exactly one choice of value. On the other hand, fail has none; (|||) adds up the choices. As was said already, integer expressions are deterministic by design. Although we have introduced concatmap (the 'bind' of the list monad) and could have likewise introduced 'return', we do not export them. In the code they are used only at specific types (namely, integer lists). It is this property that will let us later write other implementations of NDetL, which are not at all monadic.

With this NDetL implementation of NDet, the sample test – all permutations of [1;2;3] – is run as:

```
let module M = Perm(NDetL) in M.test1
⇝ [[1; 2; 3]; [2; 1; 3]; [2; 3; 1]; [1; 3; 2]; [3; 1; 2]; [3; 2; 1]]
```

Exercise 6 *Check that the identities* (1) *and* (2) *hold in this implementation.*

Exercise 7 *Consider other implementations of NDet, in terms of delimited continuations, the delimcc library, or operating system threads.*

Exercise 8 *Add yet another implementation of NDet: e.g., using the free(r) monad. Besides the depth-first search (underlying the list implementation), try to implement complete search strategies such as breadth-first search or iterative deepening.*

Exercise 9 *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?*

Exercise 10 *Generalize the NDet signature introducing* α *repr and implement this language.*

4 Advanced non-determinism: Sorting

An immediate application of list permutation is sorting: sorting, by definition, is obtaining a sorted permutation. This definition, as is, can be written down in our DSL, giving us the sorting function sort. 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 (that is, not requiring exponential time and space, in the length of the list).

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])
```

Exercise 11 *One may say that the sortedness is expressed 'meta-theoretically'. What makes one say that?*

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]
end
```

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

Exercise 12 *The slow sort is particularly slow in the shown list implementation of NDet. Why? How to speed it up?*

Exercise 13 *Implement other classical non-deterministic puzzles from the Curry example library* `http://www.informatik.uni-kiel.de/~mh/curry/examples/`

5 When Monads will not do

Although the NDet DSL is meant for non-deterministic computations, it is not as generic and expressive as it could be. For example, the NDet signature does not define the general monadic 'bind' and 'return' operations (they were not needed for the task at hand). Implementations of NDet, such as NDetL in §3, may support these operations and even use them internally – yet not offer them to the DSL programmer. The lack of generality has an upside: the NDet DSL admits implementations that do not support 'bind' and 'return' at all. This section presents three non-monadic interpretations of NDet, and explains why they are interesting and why they fall outside the conventional monadic framework.

5.1 More efficient representation

The NDet signature in §2.1 flaunts the extreme specialization: the DSL has only integers and integer lists as data types. The conspicuous lack of general lists admits however an efficient representation. Rather than the familiar linked list of cons cells (with each cell holding one list element), we may group elements in tightly packed chunks, e.g., like in Bagwell's VList [4]. A chunk can be represented as an array, or even OCaml's Bigarray.[10] The latter is particularly efficient, e.g., in avoiding GC marking. However, Bigarrays are not polymorphic: they are restricted to integers and floating-point numbers. It so happens our integer lists fit the restriction. The accompanying code shows an implementation of NDet where integer lists are (very naively, at present) represented with int Bigarray chunks. It is now an advantage that the NDet signature fails to define return and bind, because we would not have been able to support them: the present implementation deals with non-deterministic computations on integer lists only.

Again we see the general/specific trade-off: restricting the expressivity (the set of data types to operate upon) may gain a more efficient data representation.

5.2 Abstract Interpretation

The NDet DSL signature admits truly non-standard interpretations. This section describes one such example: instead of actually performing a non-deterministic computation, we estimate the number of its non-deterministic choices and the possibility of failure. This is an example of the static analysis known as abstract interpretation [10, 22]. Our example is realistic: the Kiel Curry compiler, for one, performs a similar determinism analysis in order to produce efficient code [5].

Recall, the tagless-final DSL is defined by specifying how to compositionally compute the meaning of its expressions. The 'standard' interpreter such as NDetL in §3 takes the meaning of a non-deterministic expression to be the set (to be more precise, the OCaml list) of possible values. The non-standard interpreter to be developed in this section uses a coarse, 'abstract', semantic domain, merely approximating that set. Namely, our abstraction domain here is the expression's degree of non-determinism:

type ndet_deg = {can_fail: bool; choices: lint.t}

It records the possibility of failure and the upper bound on the number of possible values: one, two, three, etc., or many. (See Fig.1 for one implementation of integers with 'many'.) An expression of degree d1 is at least as non-deterministic as an expression of degree d2 (written as d2 $\leq$ d1) iff

d2.choices $\leq$ d1.choices $\wedge$ d2.can_fail $\leq$ d1.can_fail

[10] https://caml.inria.fr/pub/docs/manual-ocaml/libref/Bigarray.html

```
module lint = struct
  type t = Int of int | Inf          (* integers with infinity *)

  let one  : t = Int 1
  let zero : t = Int 0
  let inf  : t = Inf

  let ( + ) : t → t → t = fun x y →
    match (x,y) with
    | (Int x, Int y) → Int (x+y)
    | _ → Inf

  let ( * ) : t → t → t = fun x y →
    match (x,y) with
    | (Int x, Int y) → Int (x*y)
    | _ → Inf

  let ( ≤ ) : t → t → bool = fun x y →  (* partial order *)
    match (x,y) with
    | (_,Inf) → true
    | (Inf,_) → false
    | (Int x, Int y) → x ≤ y

  let max : t → t → t = fun x y →    (* join of the semilattice *)
    match (x,y) with
    | (Int x, Int y) → Int (if x > y then x else y)
    | _              → Inf
end
```

Figure 1: A semiring/join semilattice with 'many' (inf)

In fact, our domain is not just a partial order but a join semi-lattice (common in abstract interpretation), so that we can compute the least-upper bound on the non-determinism degree for any set of expressions (for binary joins, see join in Fig.2).

Since the degree of non-determinism is estimated statically, before evaluating an expression, it is an approximation. It is an over-approximation: an expression with the estimated degree {can_fail=**true**; choices=Int 5} may in reality finish without failure, with only two possible values. Our interpreter however guarantees that the over-approximation is sound: the expression in question may have fewer than 5 possible values, but not more than 5. An expression with the (smallest) degree:

```
let det = {can_fail=false; choices=lint.one}
```

is therefore certainly deterministic. The largest degree (the maximal element of the domain) is

```
let top = {can_fail=true; choices=lint.inf}
```

It is the least informative estimate of the actual degree of non-determinism.

The abstraction domain has more structure than a mere join semi-lattice. If e1 is the non-deterministic list (expression) with at most 2 choices and e2 is the expression with at most 3 choices, we would like to

estimate that e1 ||| e2 has at most 5 choices. On the other hand, concatenating the lists e1 and e2 should have at most 6 choices of the result. We thus need additive and multiplicative operations. Whereas the former is used only in interpreting (||||), the multiplicative operation is more common. It is called merge in the code in Fig.2; incidentally, det is its unit: merge det d ≡ d for any degree d.

Figure 2 shows the complete code of the abstract interpreter, most of which has already been explained. As any other DSL interpreter, the abstract interpreter also implements the signature NDet. The meaning of an ilist_t expression is its degree of non-determinism; int_t expressions, which are always deterministic, are represented by an abstract integer AInt. The interpretation of the recursor deserves a few words. Since our abstraction domain keeps track only of the degree of nondeterminism for a list expression but not of the length of the list, the best we can do to approximate recur c n l is to find the upper bound for recur c n when applied to lists of every possible length:

$\bigsqcup_{i=0}$ recur c n $[\mathsf{AInt}]^i$

where $[\mathsf{AInt}]^i$ is the list of length i made of abstract integers. The recursive equations defining recur make it easy to compute recur c n $[\mathsf{AInt}]^{i+1}$ if recur c n $[\mathsf{AInt}]^i$ is known. All that remains is to keep joining until the result 'stabilizes'. Since we are computing an approximation of the degree of nondeterminism, we would be satisfied with an upper bound, not necessarily the least one. Therefore, we can stop the joining iteration after some number of steps, returning top if the convergence has not been achieved by then. One may organize the fixpoint computation differently: Abstract Interpretation is a vast area. The presented tagless-final framework helps us experiment with such static analyses.

Let's take a few examples of the non-determinism analyses:

```
let open NDetAbsND in  cons (int 20) (nil ||| cons (int 10) nil) ||| fail
⇝− : NDetAbsND.ilist_t =
NDetAbsND.AList {NDetAbsND.can_fail = false; choices = NDetAbsND.Int 3}
```

The result tells that the given non-deterministic list computation, if evaluated, will have at most three possible values, and it will not fail. The result of evaluating

```
let open NDetAbsND in foldr cons nil (list [1;2;3])
```

shows that the foldr expression is deterministic. The first argument of foldr (or recur) may ignore its arguments:

```
let open NDetAbsND in recur (fun _ _ → fail) (list [1] ||| list [2]) nil
⇝ − : NDetAbsND.ilist_t =
NDetAbsND.AList {NDetAbsND.can_fail = true; choices = Iint.Int 2}
```

It is easy to see from the abstract interpreter code that we never underestimate the degree of non-determinism. Thus the analysis is sound. As an example:

```
let open NDetAbsND in
foldr (fun x l → l ||| cons x l) nil (list [1;2;3])
⇝ − : NDetAbsND.ilist_t =
NDetAbsND.AList {NDetAbsND.can_fail = true; choices = NDetAbsND.Inf}
```

We can even analyze the permutation code (see §2.2)

```
let module M = Perm(NDetAbsND) in NDetAbsND.observe M.test1
```

(with the same outcome).

Finally, we should stress that we would not have been able to abstractly interpret the DSL code, had the NDet signature required the monadic operation bind, whose general signature, recall, is

```
module NDetAbsND = struct
  type ndet_deg = {can_fail: bool; choices: Iint.t}

  let det = {can_fail=false; choices=Iint.one} (* deterministic computations *)
  let top = {can_fail=true; choices=Iint.inf}

  let merge : ndet_deg → ndet_deg → ndet_deg = fun d1 d2 →
    {can_fail = d1.can_fail || d2.can_fail;
     choices = Iint.(d1.choices * d2.choices)}

  let join : ndet_deg → ndet_deg → ndet_deg = fun d1 d2 →
    {can_fail = d1.can_fail || d2.can_fail;
     choices = Iint.(max d1.choices d2.choices)}

  type int_t = AInt                    (* An abstract integer *)
  let int: int → int_t = fun _ → AInt

  type ilist_t = AList of ndet_deg
  let nil:  ilist_t = AList det
  let cons: int_t → ilist_t → ilist_t = fun _ x → x
  let list: int list → ilist_t   = fun _ → AList det

  let merge_lst : ilist_t → ilist_t → ilist_t = fun (AList d1) (AList d2) →
    AList (merge d1 d2)
  let join_lst : ilist_t → ilist_t → ilist_t = fun (AList d1) (AList d2) →
    AList (join d1 d2)

  let recur:
    (int_t * ilist_t → (unit → ilist_t) → ilist_t) → ilist_t → ilist_t →
    ilist_t = fun f z l →
      let rec loop acc res_i i =
        let res_i' = f (AInt, AList det) (fun () → res_i) in
        let acc' = join_lst acc res_i' in
        if acc = acc' then acc
        else if i > 5 then AList top
        else loop acc' res_i' (i+1)
      in merge_lst l (loop z z 0)

  let fail: ilist_t = AList {can_fail=true; choices=Iint.one}
  let (|||): ilist_t → ilist_t → ilist_t = fun (AList d1) (AList d2) →
    AList {can_fail = d1.can_fail && d2.can_fail;
           choices = Iint.(d1.choices +  d2.choices)}
end
```

Figure 2: Abstract interpreter to estimate the degree of non-determinism

```
val bind  : α m → (α → β m) → β m
```

for some parameterized type m. The second argument to bind, the continuation, is to receive the value α produced by the computation of the first bind argument. If α m is realized as ndet_deg, it can never produce any concrete α value. Therefore, when abstractly interpreting the bind expression, we cannot ever invoke, and hence analyze, its continuation. That monadic programs cannot be statically analyzed by choosing a suitable abstract monad interpretation was the main motivation for applicative functors [27] and arrows [19]. We refer to that literature for more discussion.

Exercise 14 *The recur code in Fig.2 stops the joining after 5 iterations. Explain why stopping after the first iteration would have sufficed.*

Exercise 15 *Make the analysis more precise by also tracking the size of the integer list, if known statically.*

5.3 Code Generation

This section describes yet another interpreter for the NDet DSL, which is non-standard in a different way. Rather than evaluating a DSL expression, it generates code for it. The code can be saved into a file, and then compiled and linked as any other OCaml code. The interpreter in this section is thus a DSL compiler, turning DSL expressions into ordinary OCaml code and libraries.

For code generation we rely on MetaOCaml [23], which is a superset of OCaml that adds the type α code denoting so-called code values: (fragments of) the generated code. MetaOCaml provides two primitives to build such code values. Brackets quote an OCaml expression

```
let c = .<5 + 7>.
⇝ val c : int code = .<5 + 7>.
```

turning it, *without evaluating*, into a fragment of the generated code. The *escape*, or splice, is a form of antiquotation, in Lisp terminology. It lets us build code templates with holes in them, to be later filled with other fragments, for example:

```
let template x y = .<if .~x>1 then .~y else .~y*2>. (* the template with two holes, x and y *)
⇝ val template : int code → int code → int code = <fun>

template .<read_int ()>. c   (* c was defined in the previous example *)
⇝ - : int code = .<if (Stdlib.read_int ()) > 1 then 5 + 7 else (5 + 7) * 2>.
```

Clearly, the code value (the generated code it contains) can be printed. It can also be saved into a file. The MetaOCaml home page [23] has more examples and explanations, with pointers to various tutorials.

The DSL compiler code in Fig. 3 interprets DSL expressions as code values: the fragments of code, which, when compiled and executed as part of the complete program will compute the expression values. For example, for integer DSL expressions we have:

```
type int_t = int code
let int x = .<x>.
```

We could have represented list DSL expressions likewise, as the code to compute the list of all choices:

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

The experience with abstract interpretation has taught us to analyze, to find out what we can say about the program before running it. We therefore incorporate some analysis (typically called 'binding-time

analysis' [21]) into the DSL compiler, which calls for the more elaborate semantic domain for non-deterministic list expressions:

```
type ilist_t =
    | K of int list code list
    | U of int list list code
```

It distinguishes the case of statically knowing the number of non-deterministic choices – in particular, knowing that a list expression is in fact deterministic. The literal list expression such as list [1;2;3] is clearly deterministic. We note that fact (by representing it with the K variant) and use later on in code generation (see Fig.3). The U variant of ilist_t corresponds to the statically unknown degree of non-determinism. It contains the code computing the choices at run-time. In contrast, in the K variant the choices are known statically, although the content of each choice is generally not and is to be computed at run-time. U and K hence act as annotations on the generated code – so-called *binding-time* annotations. The annotations can be erased: one may always forget the static knowledge and return the opaque int list list code value. That is the purpose of the function dyn in Fig.3. Among other uses, it extracts the result of compiling the DSL expression.

Most of the DSL compiler is derived from the list monad implementation NDetL in §3 by placing brackets and escapes at appropriate places. (The module type lift and its implementations in Lifts are provided by MetaOCaml to 'lift' OCaml values to the code that, when later run, will produce that value. Lifting is possible only for selected OCaml types.)

The recursor again needs a bit of explaining. Recall that in an expression recur c n l, l is a non-deterministic list computation. Therefore, the recur compiler in Fig.3 starts by checking what is already known about l: if it is definitely the failed computation (in which case the whole recur expression is also a failure), or if it is deterministic. In the latter case, we get recur1 to handle its only choice. In the general case, we build the code to process (again, using recur1) all the choices that l could produce, when evaluated.

For the sample Perm.test1 in §2.2 we generate the following code

```
val pcode : int list list code = .<
  let lv_10 = [1; 2; 3] in
  let rec go_11 = function
    | [] → [[]]
    | h_12::t_13 →
        Stdlib.List.concat @@
          (Stdlib.List.map
            (fun l_14 →
              let rec go_15 = function
                | [] → [[h_12]]
                | h_16::t_17 → (h_12 :: h_16 :: t_17) ::
                    (Stdlib.List.map (fun l_18 → h_16 :: l_18)
                      (go_15 t_17)) in
              go_15 l_14) (go_11 t_13)) in
  go_11 lv_10>.
```

When compiled and run, it produces the list of all permutations of the given sample list [1;2;3]. The code is surprisingly clear; one could have written something like it by hand.

Exercise 16 *Check that the identities* (1) *and* (2) *hold in this implementation as well.*

```
module NDetLCode = struct
  type int_t = int code
  let int x = .<x>.

  (* Utilities *)
  let scons : α code → α list code → α list code = fun x l →
    .<.~x :: .~l>.
  let concatmap: (α → β list) code → α list code → β list code =
    fun f l → .<List.concat (List.map .~f .~l)>.

  type ilist_t =
    | K of int list code list
    | U of int list list code

  let dyn : ilist_t → int list list code = function
    | K ls → List.fold_right scons ls .<[]>.
    | U ll → ll

  let nil = K [.<[]>.]
  let cons: int_t → ilist_t → ilist_t =
    fun x → let x = genlet x in function
      | K ll → K (List.map (scons x) ll)
      | U ll → U .<List.map (fun l → .~(scons x .<l>.)) .~ll>.
  let list x =                          (* Lifts is part of MetaOCaml *)
    let open Lifts in let module M = Lift_list(Lift_int) in
    K [M.lift x]

  let recur1: (int_t * ilist_t → (unit → ilist_t) → ilist_t) →
          ilist_t → int list code → ilist_t = fun f z l →
      U .<let rec go = function
          | []   → .~(dyn z)
          | h::t → .~(dyn @@ f (.<h>.,K [.<t>.]) (fun () → U .<go t>.))
         in go .~l>.

  let recur: (int_t * ilist_t → (unit → ilist_t) → ilist_t) → ilist_t → ilist_t → ilist_t =
    fun f z → function
      | K []  → K []
      | K [l] → recur1 f z l
      | ls    →  U (concatmap .<fun l → .~(dyn @@ recur1 f z .<l>.)>. (dyn ls))

  let fail: ilist_t = K []
  let (|||): ilist_t → ilist_t → ilist_t = fun l1 l2 → match (l1,l2) with
    | (K l1, K l2) → K (l1 @ l2)
    | (K ls, U ll)
    | (U ll, K ls) → U (List.fold_right scons ls ll)
    | (U l1, U l2) → U .<.~l1 @  .~l2>.

  let obs : ilist_t → int list list code = dyn (* Finally, the observation function *)
end
```

Figure 3: The staged DSL interpreter: the DSL compiler

Exercise 17 *Think about the ways to improve the binding-time analysis. For example, how to represent the choices that are only partially statically known? The list to process may also be (fully or partially) known statically. When unrolling recursive calls, beware of code explosion.*

Exercise 18 *How would you extend the DSL compiler to generate slowsort code, explained in §4?*

This DSL compiler would not have been possible had the NDet DSL required the monadic interface. Indeed, if it were, it would have had to support the following operations:

val return : $\alpha \to \alpha$ code
val bind : α code $\to (\alpha \to \beta$ code$) \to \beta$ code

Both of them are deeply problematic. First, not every OCaml value is convertible to the code that can be saved into a file and, when run, reproduces the value. Think of closures, reference cells and I/O channels: which code to write into a file to represent the currently open I/O channel in its current state? We are *not* saying that the types (int→int) code, int **ref** code, in_channel code, etc. are unpopulated. They clearly are, for example .<**ref** 1>. : int **ref** code. What we cannot do is to take an int **ref** value (a location in the current program heap) and convert it to code to save into a file, which, when compiled and run will yield the same location. After all, by the time the generated code is run the current program along with its heap may be long gone. The purpose of the lift module briefly mentioned earlier is to delineate the types of those values that can be converted to the corresponding code (i.e., liftable). The operation bind is likewise problematic for code values. Its second argument is a function that takes the value meant to be produced by the code supplied as the first argument to bind. The generated code, generally, cannot be run until the generation process is finished: for example, because the code may contain free variables, to be bound later in the process. Therefore, bind cannot in general apply its second argument. To put it another way, code generation cannot, generally, be influenced by the result of the already built code. All in all, bind and return with the above interface and satisfying the familiar monad laws are inexpressible.

6 Objections and Discussion

Having presented the experiment of writing effectful programs as a combination of a simple DSL embedded into a powerful metalanguage, we now discuss the results. This section concentrates on the comparison to monads and answering the commonly heard objections. §7 discusses the history and the origins of the underlying theoretical ideas.

6.1 Do we still clearly separate effectful computations?

One of the deservingly appreciated benefits of monadic programs is the clear separation of effectful computations in types and syntax. Our DSL is meant to be simple and first-order, and hence does not support the monadic interface. Yet, NDet exhibits an equally clear separation of effectful computations. Anything of the type 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 transforms non-deterministic computations.

6.2 It is not generic

Another benefit of monads is the uniformity of representing many (although not all) sorts of effectful computations and especially effectful abstractions: higher-order effectful functions. Our factoring approach is not generic. An anonymous reviewer of an early version of this paper (the extended abstract

submitted to the ML Family Workshop 2017) well described the situation and voiced the objection as follows:

> This style of programming with non-determinism seems both obvious and awkward. Everything has to be explicitly lifted and insofar as the approach is different from the traditional monadic way of structuring this kind of code, it seems less generic and less uniform (there's no guiding structure to say how higher-order functions should be lifted, for example).
>
> If so, this caveat should be pointed out, as it stands in contrast to the situation with the monadic approach, where a single definition of the monad suffices for all types.

One may quibble with the negative tone of "everything has to be explicitly lifted" assessment: after all, in the monadic approach one also has to lift all literals and the results of any pure computation. Monadic return is ubiquitous. As far as the facts of the matter are concerned, the reviewer's description is accurate. The factoring approach is not generic, is not generally applicable, and is not uniform.

Our experiment has demonstrated, however, that we did not need higher-order domain-specific functions to successfully solve the problems at hand. The experience with LMS [35] likewise shows that for many practical problems (including machine learning and data base queries), a first-order DSL is sufficient. We are not the first to observe that many typical functional programs can be written without higher-order functions (we discuss this point in more detail in §7). The first-order nature of the DSL greatly simplifies its implementation and reasoning. As to generality, even in Haskell community one begins to hear the advice "don't generalize until you use it twice" and "strive for meaningful rather than generic interfaces".

The lack of uniformity and of genericity has an upside: efficient and non-standard interpretations, as we described in §5. In particular, code generation also lacks a uniform way to lift a value to the corresponding code, see §5.3. That does not pose a problem for our approach but does for the monadic one.

6.3 But monads are 'pure'!

Finally, we cannot pass on the commonly heard slogan that the monadic code is 'pure' (and, by implication, is 'better'). Purity appeared in the Christoph Höger's message, quoted in §1.1, that prompted this paper. Purity is often used as a political slogan and rallying cry.[11] If we do look at the purity of monadic code rationally, we see nothing but confusion.

Indeed, let's look again at Christoph Höger's example, extended with an extra line for illustration:

```
do_;
  i ⟵ let f x = incr x in f 3 ;
  p ⟵ put (5 + i) ;
  j ⟵ let f x = incr x in f 3 ;
  return p
```

It may make sense to abstract the pattern:

```
let putp m = do_;
  i ⟵ m ;
  p ⟵ put (5 + i) ;
  j ⟵ m ;
```

[11] It is worth pointing out that Hughes [20, §1] noted that the standard advantages of functional programming – referential transparency, absence of side effects, no explicit control flow – is something that outsiders do not take too seriously. "Even a functional programmer should be dissatisfied with these so-called advantages," he wrote.

```
  return p
```

The original code is recovered by the instantiation putp (**let** f x = incr x **in** f 3). Suppose putp is used as a part of a bigger computation:

```
let big m = do_;
  i ⟵ m;
  j ⟵ putp m;
  return (i+1,j)
```

Since big has to evaluate its argument m anyway, one may be very tempted to 'optimize' big as

```
let bigO m = do_;
  i ⟵ m;
  j ⟵ putp (return i);
  return (i+1,j)
```

That is, we share the result of the computation rather than the computation m itself – and, inadvertently, change the behavior of our program. In this simple example, the problem is rather apparent. On one hand, one should not be too surprised: higher-order facility – just like the C preprocessor – gave us the ability to abstract computations rather than values. Functions such as big, like C macros, can be rather subtle: their seemingly straightforward refactoring often leads to subtle bugs. To be sure, this is not the problem created by monads – yet monads do little to ameliorate it. When we write effectful code – monads or no monads – we have to constantly keep in mind the context of expressions we pass around.

The fact that monadic code 'desugars' (is implementable in terms of) side-effect-free code is irrelevant. When we use monadic notation, we program within that notation – without considering what this notation desugars into. Thinking of the desugared code breaks the monadic abstraction. A side-effect-free, applicative code is normally compiled to (that is, desugars into) C or machine code. If the desugaring argument has any force, it may be applied just as well to the applicative code, leading to the conclusion that it all boils down to the machine code and hence all programming is imperative.

Like Hughes [20, §1], I object to the purity argument also methodologically. A particular programming style should be judged on its merits rather than on appeal to emotion. The merits (ease of writing, ease of implementing, code reuse among several implementations, extensibility) ideally should be evaluated by observation and experiment. Unfortunately, (properly done) empirical studies of programming styles are few and far between. From the personal experience, I have noticed that the mistakes I make when writing monadic code are exactly the mistakes I made when programming in C.[12] Actually, monadic mistakes tend to be worse, because monadic notation (compared to that of a typical imperative language) is ungainly and obscuring.

7 History and connections

We owe the main idea of the factoring approach – representing a program as a simple DSL with a powerful macro system – to Milner and his Meta Language [14]. We use the metalanguage, however, to build executable, effectful expressions rather than formulas and theorems. One may trace the origin of the approach back to Church's design of the typed lambda-calculus [8], meant to be the metalanguage providing for abstraction, definition and naming – into which one may embed a logic DSL with constants

[12] In fact, the bigO example is a very simple version of an actual problem in one of my programs. That mistake has lead to the redesign of the interface of enumerators, making it less elegant but also less error-prone.

such as equality, $\vee$, $\forall_\alpha$, etc. This idea was further developed in the Logical Framework LF [17]. ML was explicit, however, in letting programmers define their own interpretations of constants – what we have demonstrated with several different implementations of the NDet signature.

That typical higher-order functional programs can be written in a first-order language enhanced with parameterized modules (that is, endowed with a good 'macro' facility) was clearly enunciated by Goguen [13].

> "I do not consider higher order functions *harmful*, *useless*, or *unbeautiful*; but I do claim significant advantages for avoiding higher order functions whenever possible, and I claim they can be avoided quite systematically in functional programming, by using parameterized programming instead." [13, Sec. 1]

Avoiding higher-order functions, Goguen pointed out, brings simplicity and efficiency to interpreters and compilers, and, mainly, the ease of reasoning: correctness proofs can be done entirely in first-order logic. The tagless-final style we expound in the present paper may be considered an instance of Goguen's parameterized programming. We do not limit the signatures to (conventionally) algebraic (see Ex.3).

Moggi and Fagorzi [28] described monads as a tool for structuring – staging – of effectful computations. There are other ways to introduce sublanguages, such as the tagless-final style shown off in the present paper.

Robert Atkey has pointed out Reynolds' argument in [34] that Algol is the orthogonal combination of lambda-calculus and imperative programming. (Later, Abramsky and McCusker [1] described the 'imperative programming' part as an interaction with a process that implements the behavior of a storage cell.) Lambda-calculus can thus be thought of as a metalanguage, with the imperative part modeled as the following DSL (of process-interaction combinators):

```
module type STATE = sig
  type comm
  type exp
  type var

  val skip : comm
  val seq  : comm → comm → comm

  val const  : int → exp
  val add    : exp → exp → exp
  val (:=)   : var → exp → comm
  val read   : var → exp
  val while_ : exp → comm → comm
  val new_   : (var → comm) → comm
end
```

(I am very grateful to Robert Atkey for this signature and example.) On this formulation, Algol also has the 'programmable semicolon': the seq operation – as well as the programmable loop.

Harper [16, Sec.20. Modalities and Monads in Algol] argues that the distinction between 'pure' (context-independent) and effectful computations is *modal* but not *monadic*: specifically, in a so-called lax modality [31, 11].

8 Conclusions

We have described a direct alternative to the monadic encoding of effects: a bare-bone domain-specific language with effectful operations. The DSL is blended into a metalanguage such as OCaml; therefore, it can be kept tiny, with no abstraction facilities of its own, or even functions. The metalanguage, serving as an inordinarily expressive macro system, compensates. We have also argued for the principle of avoiding premature generalizations and abstractions.

We have reported only one experiment, which – combined with the related LMS experience – suggests that the DSL-metalanguage factoring approach to effectful programming 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 bigger exercise would be to re-implement the programming language Icon – another language with built-in non-determinism.

One may wonder how the history of (meta) programming might have turned out if the ML evolution had taken a different turn: kept using ML as the Meta Language as it was initially designed for, but building objects other than formulas and theorems – in particular, programs.

Acknowledgments

Extensive comments and suggestions by anonymous reviewers are greatly appreciated. I am particularly grateful to Robert Atkey for very many helpful suggestions, and for explanations of Idealized Algol. I thank Robert Harper for pointing out the lax modality and its discussion. This work was partially supported by JSPS KAKENHI Grant Number 17K00091.

References

[1] Samson Abramsky & Guy McCusker (1996): *Linearity, Sharing and State: a fully abstract game semantics for Idealized Algol with active expressions*. *Electr. Notes Theor. Comput. Sci* 3, pp. 2–14, doi:10.1016/S1571-0661(05)80398-6.

[2] Thorsten Altenkirch, James Chapman & Tarmo Uustalu (2015): *Monads need not be endofunctors*. *Logical Methods in Computer Science* 11(1), doi:10.2168/LMCS-11(1:3)2015.

[3] Robert Atkey (2009): *Parameterised Notions of Computation*. *J. Functional Programming* 19(3–4), pp. 335–376, doi:10.1145/158511.158524.

[4] Phil Bagwell (2002): *Fast Functional Lists, Hash-Lists, Deques and Variable Length Arrays*. Technical Report, EPFL. Available at `http://infoscience.epfl.ch/record/52465`.

[5] Bernd Braßel, Michael Hanus, Björn Peemöller & Fabian Reck (2011): *KiCS2: A New Compiler from Curry to Haskell*. In: *Functional and Constraint Logic Programming - 20th International Workshop, WFLP 2011, Odense, Denmark, July 19th, Proceedings*, *Lecture Notes in Computer Science* 6816, Springer, pp. 1–18, doi:10.1007/978-3-642-22531-4_1.

[6] Stanley Burris & H. P. Sankappanavar (1981): *A Course in Universal Algebra*. *Graduate Texts in Mathematics* 78, Springer, New York, doi:10.1007/978-1-4613-8130-3. Available at `http://www.math.uwaterloo.ca/~snburris/htdocs/UALG/univ-algebra2012.pdf`.

[7] Jacques Carette & Oleg Kiselyov (2011): *Multi-stage Programming with Functors and Monads: Eliminating Abstraction Overhead from Generic Code*. *Science of Computer Programming* 76(5), pp. 349–375, doi:10.1016/j.scico.2008.09.008.

[8] Alonzo Church (1940): *A Formulation of the Simple Theory of Types*. *Journal of Symbolic Logic* 5(2), pp. 56–68, doi:10.2307/2267254.

[9] Melvin E. Conway (1963): *Design of a separable transition-diagram compiler*. Commun. ACM 6(7), pp. 396–408, doi:10.1145/366663.366704.

[10] Patrick Cousot & Radhia Cousot (1977): *Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints*. In: Fourth ACM Symposium on Principles of Programming Language, Los Angeles, ACM Press, New York, pp. 238–252.

[11] Matt Fairtlough & Michael Mendler (1997): *Propositional Lax Logic*. Information and Computation 137(1), pp. 1–33, doi:10.1006/inco.1997.2627.

[12] Sebastian Fischer, Oleg Kiselyov & Chung-chieh Shan (2011): *Purely Functional Lazy Nondeterministic Programming*. Journal of Functional Programming 21(4–5), pp. 413–465, doi:10.1016/S0049-237X(08)72018-4.

[13] Joseph A. Goguen (1988): *Higher Order Functions Considered Unnecessary for Higher Order Programming*. Technical Report SRI-CSL-88-1, Computer Science Laboratory, SRI International.

[14] M. Gordon, R. Milner, L. Morris, M. Newey & C. Wadsworth (1978): *A Metalanguage for Interactive Proof in LCF*. In: Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages, ACM SIGACT-SIGPLAN, Tucson, Arizona, pp. 119–130. Available at `http://www-public.tem-tsp.eu/~gibson/Teaching/CSC4504/ReadingMaterial/GordonMMNW78.pdf`.

[15] Michael Hanus (2006): *Curry: An Integrated Functional Logic Language (Vers. 0.8.2)*. `http://www.curry-language.org`.

[16] Robert Harper (2017): *Commentary on Practical Foundations for Programming Languages (Second Edition)*. Available at `http://www.cs.cmu.edu/~rwh/pfpl/commentary.pdf`.

[17] Robert Harper, Furio Honsell & Gordon Plotkin (1993): *A Framework for Defining Logics*. Journal of the ACM 40(1), pp. 143–184, doi:10.1145/138027.138060.

[18] Paul Hudak (1996): *Building Domain-Specific Embedded Languages*. ACM Computing Surveys 28(4es), p. 196, doi:10.1145/242224.242477.

[19] J. Hughes (2000): *Generalising monads to arrows*. Science of Computer Programming 37, pp. 67–111, doi:10.1016/S0167-6423(99)00023-4.

[20] John Hughes (1989): *Why Functional Programming Matters*. The Computer Journal 32(2), pp. 98–107, doi:10.1093/comjnl/32.2.98. Available at `http://www.cse.chalmers.se/~rjmh/Papers/whyfp.html`.

[21] Neil D. Jones, Carsten K. Gomard & Peter Sestoft (1993): *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, Englewood Cliffs, NJ.

[22] Neil D. Jones & Flemming Nielson (1995): *Abstract Interpretation: Semantics-Based Tool for Program Analysis*. In Samson Abramsky, Dov M. Gabbay & Tom S. E. Maibaum, editors: Semantic Modelling, Handbook of Logic in Computer Science 4, Clarendon Press, Oxford , UK, pp. 527–636.

[23] Oleg Kiselyov: *BER MetaOCaml Home page*. Available at `http://okmij.org/ftp/ML/MetaOCaml.html`.

[24] Oleg Kiselyov (2012): *Typed Tagless Final Interpreters*. In: Proceedings of the 2010 International Spring School Conference on Generic and Indexed Programming, SSGIP'10, Springer-Verlag, Berlin, Heidelberg, pp. 130–174, doi:10.1007/978-3-642-32202-0_3.

[25] Oleg Kiselyov & Hiromi Ishii (2015): *Freer monads, more extensible effects*. In: Haskell, ACM, pp. 94–105, doi:10.1145/2804302.2804319.

[26] Oleg Kiselyov, Amr Sabry & Cameron Swords (2013): *Extensible effects: an alternative to monad transformers*. In: Haskell, ACM, pp. 59–70, doi:10.1145/2503778.2503791.

[27] Conor McBride & Ross Paterson (2008): *Applicative Programming with Effects*. J. Functional Programming 18(1), pp. 1–13, doi:10.1017/S0956796800003658.

[28] Eugenio Moggi & Sonia Fagorzi (2003): *A Monadic Multi-stage Metalanguage*. In Andrew D. Gordon, editor: Proceedings of FoSSaCS 2003: Foundations of Software Science and Computational Structures, 6th

International Conference, *LNCS* 2620, Springer, pp. 358–374. Available at `http://www.disi.unige.it/person/MoggiE/ftp/fossacs03.pdf`.

[29] Sebastien Mondet (2017): *Bioinformatics, The Typed Tagless Final Way*. Available at `https://icfp17.sigplan.org/event/ocaml-2017-papers-bioinformatics-the-typed-tagless-final-way`.

[30] P. J. Moylan (1992): *The Case against C*. Technical Report TR–EE9240, Centre for Industrial Control Science, Department of Electrical and Computer Engineering, University of Newcastle, Australia.

[31] Frank Pfenning & Rowan Davies (2001): *A judgmental reconstruction of modal logic*. *Mathematical Structures in Computer Science* 11(4), pp. 511–540, doi:10.1017/S0960129501003322.

[32] George Pólya (1945): *How to Solve It*. Princeton University Press, Princeton, NJ.

[33] Michael O. Rabin & Dana Scott (1959): *Finite Automata and Their Decision Problems*. *IBM Journal of Research and Development* 3, pp. 114–125, doi:10.1147/rd.32.0114.

[34] John C. Reynolds (1981): *The Essence of Algol*. In Jacobus Willem de Bakker & J. C. van Vliet, editors: *Algorithmic Languages*, North-Holland, Amsterdam, pp. 345–372.

[35] Tiark Rompf & Martin Odersky (2012): *Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs*. *Commun. ACM* 55(6), pp. 121–130, doi:10.1145/2184319.2184345.

[36] Tiark Rompf, Arvind K. Sujeeth, HyoukJoong Lee, Kevin J. Brown, Hassan Chafi, Martin Odersky & Kunle Olukotun (2011): *Building-Blocks for Performance Oriented DSLs*. In: *DSL*, pp. 93–117, doi:10.4204/EPTCS.66.5.

[37] Sam Staton (2013): *Instances of Computational Effects: An Algebraic Perspective*. In: *LICS*, IEEE Computer Society, p. 519, doi:10.1109/LICS.2013.58.

[38] Nikhil Swamy, Nataliya Guts, Daan Leijen & Michael Hicks (2011): *Lightweight Monadic Programming in ML*. In: *ICFP'11*, pp. 15–27, doi:10.1145/2034773.2034778.

[39] William W. Tait (1967): *Intensional Interpretations of Functionals of Finite Type I*. *Journal of Symbolic Logic* 32(2), pp. 198–212, doi:10.1007/BF01447860.

[40] Philip Wadler (1994): *Monads and Composable Continuations*. *Lisp and Symbolic Computation* 7(1), pp. 39–56, doi:10.1007/BF01019944.

[41] Mitchell Wand (1982): *Specifications, Models, and Implementations of Data Abstractions*. *Theoretical Computer Science* 20(1), pp. 3–32, doi:10.1016/0304-3975(82)90097-4.

[42] Leo White, Frédéric Bour & Jeremy Yallop (2014): *Modular implicits*. In Oleg Kiselyov & Jacques Garrigue, editors: *ML/OCaml*, *EPTCS* 198, pp. 22–63, doi:10.4204/EPTCS.198.2.

[43] Martin Wirsing (1990): *Algebraic specifications*. In J. van Leeuwen, editor: *Handbook of Theoretical Computer Science B*, Elsevier, pp. 675–788.

A Hints to selected exercises

List recursors (Ex. 1) Please try to write the function tail (obtaining the tail of a list) in terms of foldr and recur.

For more discussion of Gödel recursor and its connection with the fold on natural numbers (i.e., Church numerals) see `http://okmij.org/ftp/Computation/lambda-calc.html#p-numerals`.

Algebraic specifications (Ex. 3) To answer the question if NDet is algebraic we need the definition of algebraic specification. For ease of reference, we quote the definition from Wirsing's reference article [43, §2.1] (see also Burris and Sankappanavar's detailed course [6]). Formally, a (multi-sorted algebraic) signature Σ is a pair $\langle S, F\rangle$ where S is a set (of sorts) and F is a set (of function symbols) such that F is equipped with the mapping $type : F \to S^n \times S$ for some $n \geq 0$. The mapping, for a particular f of F is

often denoted as $f : s_1, \ldots, s_n \to s$ where $\{s, s_1, \ldots, s_n\} \subset S$. A Σ-Algebra consists of an S-sorted family of non-empty (carrier) sets $\{A_s\}_{s \in S}$ and a total function $f^A : A_{s_1}, \ldots, A_{s_n} \to A_s$ for each $f : s_1, \ldots, s_n \to s \in F$. For example, in the following OCaml code

```
module type NAT = sig
  type nat
  val zero : nat
  val succ : nat → nat
  val plus : nat → nat → nat
end
module Nat : NAT = struct
  type nat = int
  let zero = 0
  let succ x = x + 1
  let plus x y = x + y
end
```

NAT is the signature, whose set of sorts is the singleton $\{\text{nat}\}$ and function symbols are $\{\text{zero}, \text{succ}, \text{plus}\}$. In Wirsing's notation, one would write the type of plus as plus : nat, nat $\to$ nat. Nat is a NAT-algebra, whose carrier is the set of OCaml integers. See also Staton's extension of algebra formalism [37], permitting 'richer' signatures.

Laws of non-determinism (Ex. 4) Which equational laws should hold for non-deterministic computations is a rather complicated and controversial question, with no single answer. The web page `http://okmij.org/ftp/Computation/monads.html#monadplus` discusses some of the complexities.

Sortedness, meta-theoretically (Ex. 11) The exercise is an invitation to contemplate once again how the overall (sorting) computation is spread across the DSL and the metalanguage. The type of rld is particularly worth examining closely, asking oneself what do int list and ilist_t represent and what is the difference between them. See also Ex.18.

How to speed up the slowsort (Ex. 12) There is a paper about that: [12].