

A Pair of *tricks*

Oleg Kiselyov
Tohoku University
Sendai, Japan
oleg@okmij.org

Abstract

One way to ascertain the quality of the generated code – its performance, hygiene, memory safety, deadlock freedom, etc. – is to decorate it with annotations that explicate some aspect of the code: e.g., the count of particular operations, its free variables, the sequence of IO or memory allocation operations, correctness proof obligations. The annotations are built along with generating the code, and may be used to detect problems like scope extrusion or deadlock – well before the whole program is generated, let alone executed. Annotations are a worthy alternative to fancy types.

Generating code along with annotations is almost straightforward. The challenge comes from generating functions or other variable-binding forms, with the often used so-called higher-order abstract syntax (HOAS). That was the challenge posed by Olivier Danvy.

The present paper describes a new, simpler and general solution of the challenge – actually, of its general form: pairing of HOAS generators, or direct product of algebra-like structures with HOAS operations. The solution also overcomes the hitherto unsolved HOAS problem: latent effects.

CCS Concepts: • Software and its engineering → Domain specific languages; Source code generation; Procedures, functions and subroutines; Functional languages.

Keywords: code generation, annotation, delimited control, delimited dynamic binding, higher-order abstract syntax

ACM Reference Format:

Oleg Kiselyov. 2025. A Pair of *tricks*. In *Proceedings of the Workshop Dedicated to Olivier Danvy on the Occasion of His 64th Birthday (OLIVIERFEST '25)*, October 12–18, 2025, Singapore, Singapore. ACM, New York, NY, USA, 14 pages. <https://doi.org/10.1145/3759427.3760376>

1 Introduction

Partial evaluation – program generation and program transformation in general – is one of the first and long-running interests of Olivier Danvy, the area he started contributing to



This work is licensed under a Creative Commons Attribution 4.0 International License.

OLIVIERFEST '25, Singapore, Singapore

© 2025 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-2150-2/25/10

<https://doi.org/10.1145/3759427.3760376>

since joining Neil Jones' group at DIKU in 1986, and still continuing [1, 6, 7, 9, 12]. This paper is an offshoot of this stream, veering into generating code with bindings and, specifically, pairing of those generators. Along the way we pass through another of Danvy's long-running interests: continuations.

1.1 Code Annotations

The starting point is the challenge that Olivier Danvy posed to Chung-chieh Shan and the author almost two decades ago: generating code and determining the number of operations (multiplications) in it. Pure generative programming [46] by design does not provide for the examination of the generated code; therefore, to find the number of multiplications in it we have to count as we generate. One may, therefore, think of the multiplication count as an attribute, or annotation, on the code. As code fragments are combined into bigger fragments and eventually the complete program, the counts are tallied.

A more compelling example, described in Kiselyov and Imai [38, §4.1], is annotating code with the files that are opened, closed and read/written therein. As we generate the code and merge the annotations, we check right away that no file is manipulated before opening or after closing, regardless of the path an execution may take. The paper [38] expounded even more sophisticated annotations: session types, which tell of not mere reading/writing from a channel, but the sequence of reads and writes and the types of transmitted data. Such annotations let us verify the adherence to the communication protocol. Annotations thus may be as simple or as complex as needed for the verification at hand.

Annotating generating code with its free variables and so-called virtual let-bindings (which we touch upon in §2.5) has been used in MetaOCaml for a decade, for detecting scope extrusion and implementing let(rec) insertion [36].

Annotations, thus, are to reflect some aspect of the generated code, to serve as its abstraction, or approximation. In that sense, annotations are akin to types – what Pfenning called Curry-style, extrinsic types [43]. Annotations may exemplify even fancy types like typestate or session types – as argued in [38]. Unlike types, however, annotations are values, and are manipulated in a programming language rather than type system – in other words, in a language that was actually designed for programming.

1.2 Tricky Annotations

This paper takes a closer look at the process of code generation with annotations. It is actually not complicated save one

case: generating functions (or other binding forms), with the often used so-called higher-order abstract syntax (HOAS), see §4.4. It is this case that made the Danvy challenge a challenge. As we re-frame the challenge and earlier solutions, we find a more general question, and a more general answer. It is not about counting – it is about pairing of interpreters of languages with binding forms, or even more generally, direct product of algebra-like structures with HOAS operations.

Contributions.

- Description of code generation with annotations in tagless-final style (reminded in §2.1), raising the importance of the often neglected but unyielding latent effects problem of HOAS: §2.3.
- Generalization to pairing, or direct product of tagless-final interpreters: §3; §4 for the full higher-order case. With HOAS operations, tagless-final interpreters are no longer algebras, and the existence of products is uncertain. We propose a one-shot restriction §3.3 that lets us generically pair interpreters in any effectful metalanguage. Code generating interpreters are already one-shot. Meta-circular §3.4, §4.1 and abstract interpreters §4.2 can also be made one-shot.
- The one-shot restriction also solves the latent effects problem of HOAS: §4.4.

We start by setting up the framework for (heterogeneous) code generation – tagless-final style – and recasting Danvy’s challenge in it, stressing the challenging parts.

The paper uses OCaml (and in parts, MetaOCaml). Olivier has been using the languages of the ML family in many of his papers – and contributing: his functional unparsing [8] has become the SML standard formatted output facility. Since the complete code accompanies the paper, we will be showing only the salient parts, commenting on them. The code should be understandable without the detailed knowledge of OCaml; nevertheless, if necessary the reader may refer to the online manual <https://ocaml.org/manual/5.3/>.

2 Generating Code and Operation Counts

As mentioned earlier, the multiplication counting problem is the challenge posed by Olivier Danvy – at the IFIP WG 2.11 meeting in August 2007 at Dragor, Denmark, when the author met Danvy for the third time.¹ As many code-generation puzzles, it is about the power function – the ‘hello world’ of metaprogramming posed by A.P.Ershov in 1977 [14, 17].² Specifically, the problem is generating code of a function to

¹When I first met Olivier, at ICFP/GPCE 2002, he also posed a challenge: bluff combinator <http://okmij.org/ftp/Computation/lambda-calc.html#bluff>, which I solved during a session.

²Olivier’s relation with power is complicated. At one point in mid-2000s he advocated ‘down with power’ – at least that was the impression one could get, although Olivier’s actual position was quite more nuanced and well-justified. I did take the slogan to heart and used a more ‘down’, and also more realistic introductory example in [36].

compute x^n where n is statically known – tracking, at the same time, how many multiplications the generated code performs. It is a challenge: see §2.2. It was solved that very evening, in a baroque way: using MetaOCaml with two future stages. The solution was later hinted at in [23], without details. We now describe the challenge and reconstruct its solution, more perspicuously, using the tagless-final approach [4, 29, 33]. This rational reconstruction turns out a vantage point to peek at further and higher peaks: see §3 and §4.

2.1 Generating power

As advocated in [33], we start with the smallest language needed to represent the generated code – treating the language as a DSL embedded in a host language (OCaml, in this paper). DSL expressions of the type α are represented in OCaml as values of an (abstract, for now) type α exp. In the power example, DSL expressions are multiplication chains, built using

```
type  $\alpha$  exp
val one : float exp
val mul : float exp  $\rightarrow$  float exp  $\rightarrow$  float exp
```

Such a collection of operations with their types is called signature – in OCaml and in mathematics [19, §2]. Our DSL hence is an *algebra* over the that signature. It is the algebra with the constant one, and the binary operation mul to combine two expressions creating their product. The operation mul is meant to be associative with one as its unit: in other words, our algebra is a monoid.

The introduced signature already lets us write the program to build a DSL expression for x^n given an integer n :

```
let rec power (n:int) (x:float exp) : float exp =
  if n = 0 then one else mul x (power (n-1) x)
```

One may view this OCaml code as a *macro* to produce the float exp representing the product of n -many x . OCaml hence acts as a very expressive ‘macro-processor’ – as ML was originally intended [20]. This simple-minded multiplication chain is not what A.P.Ershov had in mind; he aimed for a faster, better code, as we describe later. For now, the naive power will do.

The solution is not complete: we need to generate not just a multiplication chain but a function that takes x and computes x^n . We have to add functions (procedures) to our DSL (we shall call the extended DSL signature mulproc):

```
type  $\alpha$  proc
val proc1 : (float exp  $\rightarrow$  float exp)  $\rightarrow$  (float $\rightarrow$ float) proc
```

The abstract type α proc is meant to represent a DSL function of the type α ; the operation proc1 is the generator of such one argument procedures. The type α proc is not the same as α exp: we do not assume that our DSL has first-class functions. Although proc1 is higher-order, it is a higher-order OCaml function: a higher-order macro. The DSL remains first-order.

We have, thus, been using an expressive metalanguage (macro processor) to generate code in a simple target language – the idea that motivated ML [20], advocated by Kamin [26, 27], underlies heterogeneous metaprogramming [13, 35], and, come to think of it, was championed by Goguen [18].

The DSL operation `proc1` receives an OCaml function that produces the DSL procedure body given the DSL expression representing procedure’s argument. One may say that `proc1` converts an OCaml function, on `exp` types, to a DSL function. It looks like a trick: it cannot be true in general: OCaml, and its functions, are far richer than our DSL. Using metalanguage functions to generate or represent DSL functions has been enticing and problematic for very long time [44], which we discuss in §4.4. Incidentally, with `proc1`, our DSL is no longer an algebra, strictly speaking. This is another sign of the trouble to come.

Nevertheless, this set-up lets us generate the DSL code for the function to compute x^n for a given n , say, 7.

```
let power7 = proc1 (power 7)
```

That is, we can write and type check that OCaml expression. To evaluate it and actually obtain code we need an implementation for our DSL: a concrete realization of `one`, `mul` and `proc1` operations for appropriately chosen concrete instances of the types `α exp` and `α proc`.

The first DSL realization that comes to mind is meta-circular, dissolving the DSL into OCaml. Throughout the paper, DSL implementations are presented as OCaml modules, letting us attach a name (R in our case) for ease of reference.³

```
module R = struct
  type α exp = α
  let one = Float.one
  let mul = Float.mul
  type α proc = α
  let proc1 f = f
end
```

The DSL hence is a subset of OCaml; its operations are OCaml operations.⁴ In the R implementation, `power7` then is a `float→float` OCaml function; applied to 2.0 it produces the expected 128. Such spot-testing of the generator is one of the main uses of the meta-circular implementation. In this implementation, `proc1` does convert an OCaml function to a DSL function, since they are one and the same. As we later see, in §2.3, this is exactly the problem.

More relevant to code generation is the implementation (here, OC) that represents the DSL as a quoted OCaml expression. To be precise, we are using (quasi)quotes as provided by MetaOCaml [3, 30, 36, 37].

³Float is the module in the standard library.

⁴Strictly speaking, however, a DSL expression is represented by the value it evaluates to if it were an OCaml expression. We shall see very soon why such splitting hairs is important.

```
module OC = struct
  type α exp = α code
  let one = .< 1. >.
  let mul x y = .< ~x * .~ y >.
  type α proc = α code
  let proc1 f = .<fun x → .~(f .<x>.)>.
end
```

Brackets `.<...>` are the quotation marks (similar to double-quotes used to quote strings) and `~` is the anti-quotation, or splice. MetaOCaml quotes are typed: a quoted expression of type `α` has the type `α code`. In this implementation, `power7` produces the `(float → float)` code

```
.<fun x_1 →
  x_1 *. (x_1 *. (x_1 *. (x_1 *. (x_1 *. (x_1 *. (x_1 *. (x_1 *. 1))))))>.
```

Needless to say, one may also generate code as plain strings, as the following implementation demonstrates. It takes advantage of the fact that the types `α exp` and `α proc` are distinct, as far as DSL users are concerned: Our DSL does not have to have first-class functions. Therefore, we may generate C.

```
module CC = struct
  type α exp = string
  let one = "1"
  let mul x y = Printf.sprintf "(%s*_%s)" x y
  type α proc = string
  ...
end
```

In general, C generation is quite more involved: see [35]. In the CC implementation, `power7` evaluates to the string

```
float pw7(float const x1) {
  return (x1 *(x1 *(x1 *(x1 *(x1 *(x1 *(x1 *(x1 * 1))))));
}
```

2.2 Counting Problem

Now that we can generate the power function, we move to the main problem: generate code and report how many multiplications it does when executed. It seems trivial: we merely need to adjust the generator to literally count each multiplication as it is generated:

```
let cnt = ref 0
let rec power' (n:int) (x:float exp) : float exp =
  if n = 0 then one else (incr cnt; mul x (power' (n-1) x))
let power7' = let c = proc1 (power' 7) in (c,!cnt)
```

With the OC implementation of the DSL, we indeed obtain the generated code paired with the multiplication count: 7.

There are three problems with this solution. First, we had to fiddle with the power generator and edit its code. The generator may have been provided by a third party; meddling with its source code (if available at all) could be cumbersome, let alone error-prone. Second, the approach is non-modular,

or non-compositional: rather than annotating each code fragment with its multiplication count (tallying the counts as the fragments are composed), power' counts multiplications in an out-of-bound way, for the power generator as a whole. To clearly see the problem, let's consider a better algorithm of computing x^n : iterated squaring, which is the algorithm that Ershov was talking about [17].

```
let rec power2 (n:int) (x:float exp) : float exp =
  match n with
  | 0 → one
  | 1 → x
  | n when n mod 2 = 0 → power2 (n / 2) (mul x x)
  | n → mul x (power2 (n-1) x)
```

Similarly to power', one could add incr cnt before mul. The power₂ 7 adjusted for counting then generates, with the OC implementation

```
fun x1 → x1 *. ((x1 *. x1) *. ((x1 *. x1) *. (x1 *. x1)))
```

and reports the multiplication count 4. Not only we still get the naive linear multiplication chain; the count of its multiplications is wrong.

Finally, if we evaluate power7' with the meta-circular R implementation, the reported multiplication count is zero. The reader may wish to pause to think why the counting fails so spectacularly.

2.3 Towards Compositional Counting

Let us change the strategy: rather than modifying a generator like power, let's modify a DSL implementation, to annotate each generated code (fragment) with the count of its multiplications. This strategy solves at least two of the problems noted in §2.2: keeping the generator as is, and making sure the count always corresponds to the code, by construction. §2.1 defined three different ways to represent the code: three different implementations of the DSL; more can always be added. Manually modifying each of them to count multiplications is bothersome. More satisfying is transforming a DSL implementation into a counting one. In OCaml, we arranged DSL implementations to be modules; a module transformer then is what OCaml calls a functor:

```
module Counter(L:mulproc) :
  (mulproc with type  $\alpha$  proc =  $\alpha$  L.proc * int) =
  struct
    type  $\alpha$  exp =  $\alpha$  L.exp * int
    let one = (L.one, 0)
    let mul (xc,xn) (yc,yn) = (L.mul xc yc, xn + yn + 1)
    type  $\alpha$  proc =  $\alpha$  L.proc * int
    let proc1 = (* see text *)
  end
```

It takes a module (a DSL implementation) of the signature mulproc (see §2.1) to be referred to as L, and produces another DSL implementation, with the α proc being realized as

the pair of L's α proc and an integer: the multiplication count. As we see, Counter(L) pairs the generated code (both the top-level α L.proc and the general α L.exp) with the count of multiplications therein. As code is built/combined, the counts are tallied up. The counting is compositional: it depends only on the operation that combines code fragments and the counts of the fragments.

Implementing proc1 to deal with the count annotations is tricky. We would like it to look along the lines of

```
let proc1 (f: float exp → float exp) : (float → float) proc =
  let (c,n) = f (xc,0) in
  let c = L.proc1 (fun xc → c) in
  (c,n)
```

where xc denotes the code for the argument of the generated function; it is annotated with the multiplication count of 0 since it is a value. This implementation cannot be right: the variable xc is used before bound. We have no choice but to move the application f (xc,0) inside:

```
let proc1 f =
  let c = L.proc1 (fun xc → let (c,n) = f (xc,0) in c) in
  (c,n)
```

which creates another problem: getting n (the count of multiplications in the function's body) out. The only way around is by a side-effect, it seems:

```
let proc1 f =
  let nr = ref 0 in
  let c = L.proc1 (fun xc → let (c,n) = f (xc,0) in nr := n; c) in
  (c, !nr)
```

At first this seems to work, with the DSL implementation Counter(OC) (that is, the implementation OC transformed by Counter): the power7 generator of §2.1, as is, generates the code, seen earlier, and reports the multiplication count 7. The optimized power₂ 7 generator from §2.2 produces

```
fun x1 → x1 *. ((x1 *. x1) *. ((x1 *. x1) *. (x1 *. x1)))
```

and reports the count 6. Although the code is still the naive linear multiplication chain, at least the multiplication count is correct now.

However, the Counter(R) implementation still reports the multiplication count of zero. It is instructive to understand why. In the R implementation, see §2.1, proc1 is the identity function. Therefore, in Counter(R), the operation proc1 becomes:

```
let proc1 f =
  let nr = ref 0 in
  let c = fun xc → let (c,n) = f (xc,0) in nr := n; c in
  (c, !nr)
```

Here, c is the generated proc function and f is the generator of its body (which also counts the multiplications as they are generated). The generation of the body f (xc,0) is not evaluated until c is applied. Therefore, at the time c is produced, no counting is performed.

We have run into the common metaprogramming trouble: a computation that we thought is done early, at the time of specialization/code generation, is in fact performed later, when the generated code is run. Looking back, our first attempt at `proc1` has the right order of first generating the function body and then generating the whole function – but had the problem with binding. The final attempt gets the binding right, but, in the case of R implementation, the generation order is odd. That the order of evaluation (of generators) does not follow the order of binding (for variables in the generated code) is a salient, and subtle feature of meta-programming, expounded in [23].

Undoubtedly, Olivier knew of all these problems: that is why he posed the challenge to design the proper generator, that correctly annotates code with the multiplication count no matter how that code might be represented.

2.4 Solving the Challenge

The solution developed back at the Dragor 2007 meeting was to annotate, so to speak, the generated code not just with the count of its multiplications but also with its free variables. For our simple problem the free variable annotation is simple. Since the DSL has just a single one-argument procedure, there can only be one free variable: that procedure’s argument (of the type `float exp`). Therefore, DSL expressions are represented as `fun v → ...` where `v` is the value to associate to that argument – in other words, as functions from the environment. The solution is as follows:⁵

```
module Counter(L:mulproc) :
  (mulproc with type  $\alpha$  proc =  $\alpha$  L.proc * int) =
struct
  type  $\alpha$  exp = (float L.exp →  $\alpha$  L.exp) * int
  let one = (Fun.const L.one, 0)
  let mul (xc,xn) (yc,yn) =
    ((fun env → L.mul (xc env) (yc env)), xn + yn + 1)
  type  $\alpha$  proc =  $\alpha$  L.proc * int
  let proc1 (f:float exp → float exp) : (float→float) proc =
    let (cenv,n) = f (Fun.id,0) in
    let c = L.proc1 cenv in
    (c,n)
end
```

Recall the first, discarded attempt to write `proc1` in §2.3: it had the right order of first generating the `proc` body and then the procedure itself – but represented the yet to be bound procedure’s argument as a yet to be bound OCaml variable. OCaml, however, requires all variables be bound before use and does not allow evaluating open code. Our solution gets around the problem: the free, from the DSL point of view, variable is represented as the closed OCaml value: `fun v → v` (the function that takes the environment and extracts the value associated with the variable). Now,

⁵Fun is a module in the standard library.

`proc1` in `Counter(L)` always generates the body of the procedure before the procedure itself – for any `L`, including `R`. The counting now works in every DSL implementation.

We have presented the rationalized, with the benefit of hindsight, version of the original answer to Danvy challenge. Its key idea, of free variable annotations, has been developed extensively since the Dragor meeting [23, 25, 34, 36, 39].

2.5 Power by Squaring

Although a digression, we cannot avoid describing an efficient power generation – after all, it also has a connection with Danvy’s research, and with annotations. The Ershov algorithm was already mentioned (repeated below):

```
let rec power2 (n:int) (x:float exp) : float exp =
  match n with
  | 0 → one
  | 1 → x
  | n when n mod 2 = 0 → power2 (n / 2) (mul x x)
  | n → mul x (power2 (n-1) x)
```

Since `power2 n` requires fewer recursive calls than `power n`, the code generation becomes faster. The generated code, however, as seen in §2.3 for `n = 7`:

```
fun x1 → x1 *. ((x1 *. x1) *. ((x1 *. x1) *. (x1 *. x1)))
```

still has almost as many multiplications as the naive power algorithm. The reason is that `power2 n (x:float exp)` is a macro, expanding into a series of multiplications of its argument `x`, which is hence replicated in the result. If that argument happens to be a non-trivial expression, it is likewise replicated. That macros are prone to duplicate code is the folklore, spelled out explicitly in e.g., [28, 45].

The way to avoid duplication is sharing, and the way to express sharing in code is by introducing temporary variables to name subexpressions and hold their results – in other words, by let-binding.^{6 7} For example, MetaOCaml has the dedicated let-binding facility [36], in particular, providing

```
val genlet :  $\alpha$  code →  $\alpha$  code
```

One may view it as an annotation (pun intended) to let-bind a code fragment at an appropriate place; the result of `genlet` is the code of the variable let-bound to the expression.

For example, if we replace the implementation of `mul` in the OC implementation with

```
let mul x y = genlet .< .~x *. .~ y >.
```

then the naive `power7` produces

```
fun x1 →
```

⁶The same Ershov has pioneered the approach to detect common sub-expressions and eliminate them via sharing – the approach that is now called ‘hash-consing’ [15, 16]. Ershov’s paper was however published before Lisp (and hence the name `cons`) existed.

⁷Tseitlin algorithm for representing a boolean expression (combinational circuit) in conjunctive normal form is another example of using sharing to avoid code duplication, by introducing auxiliary variables to name sub-expressions.

```
let t_2 = x_1 *. 1.      in let t_3 = x_1 *. t_2 in ...
let t_8 = x_1 *. t_7    in t_8
```

The repeated squaring `power27` generates

```
fun x_1 →
  let t_2 = x_1 *. x_1    in let t_3 = t_2 *. t_2 in
  let t_4 = t_2 *. t_3    in let t_5 = x_1 *. t_4 in t_5
```

The reader is encouraged to think of a way to eliminate the last, unnecessary let-binding in the generated code above.

Let-insertion is a very old topic [21], to which Danvy has also contributed [41]. Let-insertion is connected to continuations, either explicit [2, 25] or reified, via delimited control [24, 41] – which is how MetaOCaml’s `genlet` used to be implemented. About a decade ago, however, `genlet` has been re-implemented, using code annotations, or so-called ‘virtual let-binding’ [36]. Below we illustrate this technique by providing sharing in the CC implementation: the simplistic C code generator, which has no built-in `genlet`.

Whereas in the original CC implementation α exp was realized as a string of C code, it is now annotated. The annotation is `vlets`: a sequence of ‘virtual’ let-bindings, associations of names and the expressions (C code) they are (to be) bound to. We call `vlets` virtual bindings because they are not yet realized as actual C binding statements; the realization occurs in `proc1`, when the top-level function is generated. For now, the virtual bindings are carried along with the expression that may use their bound variables. When expressions are combined (as in `mul`), the bindings are merged, eliminating duplicates.⁸ This elimination, a form of hash-consing, is what accomplishes sharing.⁹

```
module CCshare = struct
  type name = string          (* variable name *)
  type ccode = string        (* C code *)
  type vlets = (name * ccode) list (* virtual bindings *)
  type  $\alpha$  exp = ccode * vlets
  let empty = []             (* vlet is a monoid *)
  let rec merge (l:vlets) (r:vlets) : vlets = match (l,r) with
  | ([],x) | (x,[]) → x
  | (l,(n_)::t) when assoc_opt n l ≠ None → merge l t
  | (l,h:::t) → merge (l @ [h]) t
  let genlet ((c,vl): $\alpha$  exp) :  $\alpha$  exp =
    let v = gensym "t" in (v,vl @ [(v,c)])
  let one = ("1",empty)
  let mul (xc,xvl) (yc,yvl) =
    (sprintf "(%s_.*%s)" xc yc, merge xvl yvl)  $\triangleright$  genlet
  ...
end
```

⁸A virtually-bound expression may contain virtually-bound variables. Therefore, the order of virtual bindings in `vlets` is partially important. The merge operation preserves the partial order.

⁹ \triangleright is the left-associative swapped application operator. The OCaml standard library also provides the right-associative low-precedence application operator `@@`, which we will see later. Hence, `x + 1 \triangleright f` is the same as `f @@ x + 1` and is the same as `f (x + 1)` but avoids the parentheses.

With such `vlet` annotations, `genlet` becomes easily implementable: if we think of an DSL expression `e`, realized as `(c, [(v1,c1);(vn,cn)])`, as denoting the C code

```
float v1 = c1; float vn = cn; ... c
```

`genlet e` then corresponds to

```
float v1 = c1; float vn = cn; float v = c; ... v
```

In this CCshare implementation, `power27` generates

```
float foo(float const x1) {
  float const t2 = (x1 * x1);
  float const t3 = (t2 * t2);
  float const t4 = (t2 * t3);
  float const t5 = (x1 * t4);
  return t5;
}
```

3 Pairing of Implementations

The answer to Danvy challenge in §2.4 is specifically tailored to the case of generating a single second-class function of a single floating-point argument. Generalizing is not simple: the underlying approach of annotating code with free variables [23] is complicated, requiring first-class records with row-polymorphism and at times explicit manual environment adjustments. In the present and the following sections we look at the challenge from a different point of view and discover a more general question, and a more general answer – which ensures meta-circular interpreters no longer cause trouble. The present section introduces the different point of view, still in the context of the power-like example. §4 extends to the higher-order case.

3.1 Abstract Interpretation

Danvy’s challenge and its original solutions reviewed in §2 were in the context of the classical partial evaluation or its explicit form, staging. The present paper, however, frames the problem in the tagless-final style. Its characteristic is multiple interpretations: the same DSL expression (such as `power7`) may be interpreted with many interpreters, such as `R`, `OC`, `CC` – as we have demonstrated earlier. The point of view of multiple interpretations lets us see another way to count multiplications: by writing a counting interpreter, which interprets DSL expressions as the count of the multiplications therein.

```
module N = struct
  type  $\alpha$  exp = int
  let one = 0
  let mul x y = x + y + 1
  type  $\alpha$  proc = int
  let proc1 (f:float exp → float exp) : (float→float) proc =
    f 0
  end
```

When interpreting `proc1`, we rely on its being first-order: its

argument is a base-type value, and so is its result. Neither includes any latent computations such as closures. The general case is discussed in §4.2.

The implementation N does a compositional, *abstract interpretation* as expounded in [22]. The abstract interpretation here is trivial because the DSL is: first-order with no iteration or recursion, or even function application. In the N implementation, power_7 from §2.1 gives 7, and $\text{power}_2 7$ from §2.2 gives the expected 6 as the multiplication count.¹⁰

3.2 Direct Product of Implementations

The abstract interpretation N does the counting, and the OC or CC implementations of the DSL do the code generation. On the other hand, the code annotation approach in §2.3 counts multiplications as it generates the code: the generation and counting go in lock-step. The two approaches are closely related, however: generating of the count-annotated code may be regarded as pairing of the code generation with the abstract interpretation – or, in other words, as the direct product of two DSL implementations.

Pairing of two DSL implementations, L and R , is as simple (to a point) as it sounds:

```
module BProduct(L:mulproc)(R:mulproc) :
  (mulproc with type  $\alpha$  proc =  $\alpha$  L.proc *  $\alpha$  R.proc) =
  struct
    type  $\alpha$  exp =  $\alpha$  L.exp *  $\alpha$  R.exp
    let one = (L.one, R.one)
    let mul (xl,xr) (yl,yr) = (L.mul xl yl, R.mul xr yr)
    type  $\alpha$  proc =  $\alpha$  L.proc *  $\alpha$  R.proc
    (* let proc1 (f: float exp → float exp) = ??? see §3.3 *)
  end
```

So far it is the textbook-standard direct product of algebras. However, proc1 is a challenge – exactly the same challenge we faced in §2.3. The reason is that proc1 is not an algebraic operation.

Assuming the product proc1 is implementable, a Counter-annotated DSL implementation may be formally related to the abstract interpretation:

$$\text{Counter}(X) = \text{BProduct}(X)(N)$$

for any DSL implementation X . Incidentally, this equality gives another way to implement Counter. For the algebraic fragment of the DSL, the proof is self-evident, just by looking at the Counter and BProduct code. The complicated part is proc1 , to which we turn now.

3.3 One-Shot Generation

For the hint on proc1 for the product, let us look at its implementation as part of Counter(L) in §2.3, repeated below.

¹⁰The implementation N , as an abstraction of OC and CC implementations does not account for sharing. The reader is encouraged to try to adjust N so to count multiplications in the generated code with sharing.

Recall, that α exp was represented as the pair of α L.exp and an integer, the count.

```
type  $\alpha$  exp =  $\alpha$  L.exp * int
let proc1 (f: float exp → float exp) =
  let nr = ref 0 in
  let c = L.proc1 (fun xc → let (c,n) = f (xc,0) in nr := n; c) in
  (c, !nr)
```

The implementation relied on a side-effect (mutation, although we could have used coroutine or delimited control) to smuggle-out the annotation n from outside L.proc1. It worked when L is the OC or CC – but not for the meta-circular R . To understand why, let’s examine L.proc1 code in these implementations. In CC:

```
let proc1 (f: float exp → float exp) =
  let v = gensym "x" in
  Printf.sprintf "float_foo(float_const_%s)_{\n_return_%s;\n}"
  v (f v)
```

proc1 first builds the code for the to be bound variable, then evaluates $f v$ to generate the function body, and, finally, generates the binder for the variable. As the consequence, evaluating $\text{proc1} f$ applies f , exactly once. The OC implementation works similarly. On the other hand, in the metacircular R implementation,

```
let proc1 (f: float exp → float exp) = f
```

evaluating $\text{proc1} f$ merely returns f without applying it. That is the crucial difference. The side-effecting Counter(L). proc1 depended, for the proper sequencing of the side-effects, on L.proc1 applying its argument (the generator for the procedure body) exactly once. OC.proc1 and CC.proc1 do that, but R.proc1 does not.

In §2.3, we blamed the side-effecting Counter(L). proc1 and searched for the one that works for any DSL implementation including R . Perhaps the blame was misplaced: it is the R implementation that is the odd one out, failing to generate the body of the function at the time the function code is being built. Perhaps we should have rejected not proc1 but R , imposing the following restriction.¹¹

Definition 3.1 (One-shot function generation restriction). A function generator $(\alpha \text{ exp} \rightarrow \beta \text{ exp}) \rightarrow (\alpha \rightarrow \beta)$ proc before returning the output function $(\alpha \rightarrow \beta)$ proc must apply the received function body generator $\alpha \text{ exp} \rightarrow \beta \text{ exp}$ exactly once.

One may notice the similarity with “one-shot” continuations such as those associated with effects in the OCaml 5.x: one-shot captured continuations must be invoked, exactly once. Strictly speaking, the function generator should be ascribed a type like

$$1:(\alpha \text{ exp} \rightarrow \beta \text{ exp}) \rightarrow \omega:(\alpha \rightarrow \beta) \text{ proc}$$

¹¹§3.4 describes the replacement of R that satisfies the restriction.

in a made-up notation¹² where 1: signifies the argument must be used once. The type system of OCaml, however, does not provide for such annotations. Therefore, the one-shot generator restriction, like the one-shot continuation restriction, has to be stated and verified “out-of-band”.

With the one-shot premise, BProduct(L)(R).proc1 is implementable similarly to the side-effecting Counter(L).proc1:

```

module BProduct(L:mulproc)(R:mulproc) :
  (mulproc with type  $\alpha$  proc =  $\alpha$  L.proc *  $\alpha$  R.proc) =
  struct
    type  $\alpha$  exp =  $\alpha$  L.exp *  $\alpha$  R.exp
    ...
    type  $\alpha$  proc =  $\alpha$  L.proc *  $\alpha$  R.proc
    let proc1 (f:float exp  $\rightarrow$  float exp) : (float $\rightarrow$ float) proc =
      let prref = ref None in
      let pl =
        L.proc1 (fun xl  $\rightarrow$ 
          let ylref = ref None in
            let pr =
              R.proc1 (fun xr  $\rightarrow$ 
                let (yl,yr) = f (xl,xr) in
                  ylref := Some yl; yr) in
                prref := Some pr;
              Option.get !ylref) in
          (pl, Option.get !prref)

```

The correctness relies on the fact that ylref (the body of the function for the L generator) and prref (for the function constructed by the R generator) are single-assignment reference cells. That fact is justified by L and R implementations obeying the one-shot restriction. The product proc1 then, too, satisfies the restriction.

More elegantly, and also more bewilderingly, proc1 may be implemented using delimited control – to which Danvy dedicated a significant part of his career, starting with his and Filinski’s seminal paper on shift [10, 11]. The code below uses a related shift0 control operator, but with a more convenient interface: so-called algebraic effects, natively supported in the recent versions of OCaml. The code below closely follows the example in the OCaml manual;¹³ **perform** *Eff* is akin to **raise** *Exc*; **try** *e* **with effect** *Eff*, *k* \rightarrow ... is akin to the exception handling **try** *e* **with** *Exc* \rightarrow ... However, the ‘exception’ *Eff* has to be resumed (“recovered from”) by reinstating the captured continuation *k*, using the `Deep.continue` resumption operator. The continuation *k* must be reinstated exactly once – which the code below relies on and ensures.

```

let proc1 (f:float exp  $\rightarrow$  float exp) : (float $\rightarrow$ float) proc =
  let open Effect in

```

```

let open struct
  type _ eff +=      (* defining new effects L and R *)
    | L : float L.exp  $\rightarrow$  float L.exp eff
    | R : float R.exp  $\rightarrow$  float R.exp eff
  exception Done of (float $\rightarrow$ float) proc
  let execute :  $\alpha$   $\rightarrow$   $\beta$  = fun _  $\rightarrow$  assert false
end in
try
  execute @@ try L.proc1 (fun xl  $\rightarrow$  perform (L xl))
  with effect L xl, kl  $\rightarrow$ 
    execute @@ try R.proc1 (fun xr  $\rightarrow$  perform (R xr))
  with effect R xr, kr  $\rightarrow$ 
    let (yl,yr) = f (xl,xr) in
    let pl = Deep.continue kl yl in
    let pr = Deep.continue kr yr in
    raise (Done (pl,pr))
  with Done x  $\rightarrow$  x

```

What is the purpose of the strange function `execute : $\alpha \rightarrow \beta$` and how it all could possibly work is left as an exercise to the reader.

3.4 One-Shot Meta-circular Implementation

The implementation R in §2.1 was simple but did not obey the one-shot restriction. We now describe the metacircular implementation that does.

```

module RL = struct
  type  $\alpha$  exp = unit  $\rightarrow$   $\alpha$ 
  let one = fun ()  $\rightarrow$  Float.one
  let mul x y = fun ()  $\rightarrow$  Float.mul (x ()) (y ())
  type  $\alpha$  proc =  $\alpha$ 
  let proc1 f =
    let r = ref 0. in
    let body = f (fun ()  $\rightarrow$  !r) in
    fun x  $\rightarrow$  r := x; body ()
  end

```

A DSL expression of type α is now represented in OCaml as a thunk rather than a value. That is already an improvement: the R implementation cannot account for non-terminating DSL computations or conditionals (in an appropriately extended DSL) but RL can.

The implementation of proc1 now satisfies the one-shot restriction: the generator of the body (the argument of proc1) is applied exactly once. The implementation relies on the fact that the function takes the value of the base type (float) and returns the value of the base type: there are no closures or latent computations involved. In the general case, we have to use delimited dynamic binding extended with the operation to capture the dynamic environment [31].¹⁴

¹²Introduced by Edsko de Vries in his article “Linearity in Haskell” <https://web.archive.org/web/20231203065819/http://www.edsko.net/2017/01/08/linearity-in-haskell/>

¹³<https://ocaml.org/manual/5.3/effects.html#%3Aeffects-basics>

¹⁴See also <https://okmij.org/ftp/Computation/having-effect.html> and <https://okmij.org/ftp/continuations/stack-env.html>

4 Pairing First-Class Function Generators

The power example belabored so far was written in a DSL with a single top-level (second class) function of a base-type argument. The DSL implementations took advantage of this simplicity. This section turns to the general case: a higher-order DSL, with first-class functions of arbitrary order (that is, whose arguments may themselves be functions). The task is to build the product (pairing) of such higher-order DSLs.

For concreteness, we take a simply-typed lambda calculus with the float base type and literals and operations on that type, represented in OCaml by the following signature:

```
module type ho = sig
  type  $\alpha$  exp
  val float : float  $\rightarrow$  float exp
  val add : float exp  $\rightarrow$  float exp  $\rightarrow$  float exp
  val mul : float exp  $\rightarrow$  float exp  $\rightarrow$  float exp
  val lam : ( $\alpha$  exp  $\rightarrow$   $\beta$  exp)  $\rightarrow$  ( $\alpha \rightarrow \beta$ ) exp
  val ( $\cdot$ ) : ( $\alpha \rightarrow \beta$ ) exp  $\rightarrow$  ( $\alpha$  exp  $\rightarrow$   $\beta$  exp)
end
```

As in §2.1, the OCaml type α exp denotes DSL expressions of type α , which are build from floating-point literals, addition and multiplication. Unlike §2.1, we no longer distinguish functions: they are also expressions and hence may be passed as arguments and returned as results. DSL functions are created using lam and applied via the left-associative infix \cdot operator (enclosed in parentheses when used as an identifier). Here are a few examples of the ho DSL expressions, to be used later:

```
let sqr = lam (fun x  $\rightarrow$  mul x x)
let smul = lam (fun x  $\rightarrow$  lam (fun y  $\rightarrow$  sqr  $\cdot$  (mul y x)))
let hof = lam @@ fun f  $\rightarrow$  f  $\cdot$  (f  $\cdot$  float 4.)
let ex2 = lam @@ fun x  $\rightarrow$ 
  let+ f1 = smul  $\cdot$  (mul x (float 5.)) in hof  $\cdot$  f1
```

For convenience we introduced let-expressions in our DSL, to name sub-expressions – taking advantage of a user-definable let-binding-like construct:¹⁵

```
let (let+) :  $\alpha$  exp  $\rightarrow$  ( $\alpha$  exp  $\rightarrow$   $\beta$  exp)  $\rightarrow$   $\beta$  exp = fun e k  $\rightarrow$ 
  lam k  $\cdot$  e
```

In a higher-order language, let-expressions do not have to be built-in: they may be defined as a macro. In the above examples, whereas OCaml's let names DSL macros, let+ acts as a let-expression in the DSL itself: unlike macros, it is not inlined. (let+ does look like an OCaml let-expression – which is the point of the user-definable let-operators).

Like the mulproc DSL earlier, ho can be implemented in several ways: as the (naive) metacircular interpreter R (we show only the higher-order fragment)

```
module R = struct
  type  $\alpha$  exp =  $\alpha$ 
  ...
```

¹⁵<https://ocaml.org/manual/5.3/bindingops.html>

```
let lam f = f
let ( $\cdot$ ) = (@@)
end
and the OCaml code generator
module OC = struct
  type  $\alpha$  exp =  $\alpha$  code
  ...
let lam f =  $\cdot$ .(fun x  $\rightarrow$   $\cdot$ .(f  $\cdot$ .(x))>).
let ( $\cdot$ ) = fun f x  $\rightarrow$   $\cdot$ .( $\cdot$ .(f  $\cdot$  x) >).
end
```

The significant difference between them is illustrated by the following example, borrowed from [23]:

```
let ex51 =
  let rapp x f = print_endline "rapp"; f  $\cdot$  x in
  lam @@ fun x  $\rightarrow$ 
    rapp (add x (float 1.))
    (lam @@ fun y  $\rightarrow$ 
      rapp (add x y) (lam @@ fun z  $\rightarrow$  mul z y))
```

Here rapp is a reverse application macro, which also prints "rapp" when executed. In the OC implementation, evaluating ex51 prints "rapp" twice and produces the code

```
fun x_1  $\rightarrow$  (fun x_2  $\rightarrow$  (fun x_3  $\rightarrow$  x_3 * x_2) (x_1 + x_2))
  (x_1 + 1.)
```

In contrast, in the R implementation, ex51 evaluates to an OCaml function while printing nothing. Whereas OC acts as a code generator, R does not: the metacircular interpreter does not separate the DSL code generation from the execution of the generated code. In other words, R does not obey the one-shot restriction (Defn. 3.1).

4.1 One-Shot Higher-Order Meta-circular Interpreter

The question of a one-shot meta-circular interpreter, separating the stages of generation and execution of the generated code, arises again. It has been answered (under some restrictions) in [23], but in a convoluted way, whose correctness is non-obvious (the paper [23] proved the type safety; the dynamic semantic correctness was only conjectured). Since then the paper [31] proposed a much simpler implementation whose correctness is easy to see. It relied on an extended version of delimited dynamic binding [40]. Below we recast the solution using explicit delimited control – or, one-shot algebraic effects of OCaml.

As in §3.4, we represent DSL expressions of type α as OCaml thunks $\text{unit} \rightarrow \alpha$. The first-order fragment is the same as in §3.4; even application is simple to implement. The main problem is the function generator lam. With only top-level second-class functions, the distinction between dynamic and lexical scoping is immaterial. Therefore, in §3.4 we were free to use dynamic binding to access function arguments. For ho, however, if it is to be a lambda-calculus, we have to implement lexical scoping, which is now distinct from dynamic

one. It has been discovered, however, that dynamic binding implements lexical binding, given a facility to capture the current dynamic environment [32].¹⁶ The following code uses effects both for dynamic binding and for capturing.

```

module RL = struct
  type  $\alpha$  exp = unit  $\rightarrow$   $\alpha$ 
  ...
  let ( $\cdot$ ) x y = fun ()  $\rightarrow$  (@@) (x ()) (y ())
  type env = {ev :  $\alpha$ . (unit $\rightarrow$  $\alpha$ )  $\rightarrow$   $\alpha$ }
  let empty = {ev = fun f  $\rightarrow$  f ()}
  type _ eff += Env : env eff

  let lam : type a b. (a exp  $\rightarrow$  b exp)  $\rightarrow$  (a $\rightarrow$ b) exp = fun f  $\rightarrow$ 
    (* generating lambda code *)
    let open struct type _ eff += Var : a eff end in
    let body = f (fun ()  $\rightarrow$  perform Var) in
    fun ()  $\rightarrow$  (* executing lambda *)
    let parent_env =
      try perform Env with Unhandled _  $\rightarrow$  empty in
    fun x  $\rightarrow$ 
      (* applying lambda *)
      let env = {ev = fun f  $\rightarrow$ 
        try parent_env.ev f with
          effect Var, k  $\rightarrow$  Deep.continue k x} in
      try env.ev body with
        effect Env, k  $\rightarrow$  Deep.continue k env
    end

```

A free variable is truly an effect: specifically, the effect Var to obtain the value associated with the variable in the current *dynamic* environment. (The effect label Var is local to each lam and hence represents its bound variable.) The type env represents the captured dynamic environment. It is, effectively, the composition of the handlers for the Var effects.

There are three points to note in the lam implementation above. The first is ‘generating lambda code’: accepting $f:(a \text{ exp} \rightarrow b \text{ exp})$ and applying it to generate the DSL expression for the lam body. Clearly, f is applied exactly once: the one-shot restriction is satisfied. The point ‘executing lambda’ marks the code for the produced DSL lambda-expression. That expression, when executed, first performs the effect Env to obtain the representation of the current dynamic environment. Hence, ‘lambda is an effect’ – the slogan of [32]. On the other hand, ‘applying lambda’ is the point when the closure created by the DSL lambda-expression is applied. At this point the captured env is extended with the new binding, and the body is executed in this extended *captured dynamic* environment. The body is executed in the environment of the closure creation rather than the environment of the application – the lexical scope, by its very definition. When the function body is executed, Env effect is handled by providing the representation of the current env.

¹⁶See also <https://okmij.org/ftp/continuations/stack-env.html>

We have demonstrated the meta-circular implementation of the ho DSL that is obviously one-shot, and is also obviously correct, being a straightforward realization of lexical scoping. One may observe that instead of *reify*ing the current environment, as env, we build env and then *reflect* it.

4.2 Typed Annotations

In the general higher-order case, multiplication counting also becomes more complicated. In the first-order case it was enough to associate with each expression the count of its multiplications. The abstract interpreter N from §3.1 interpreted DSL expressions as such multiplication counts. Values, such as numeric literals and also base-type function arguments, do no computations and hence receive the multiplication count of zero.

In the higher-order case, function values are also values; however, they contain latent computations, including multiplications, performed when the function is applied (which may be more than once). Therefore, we have to distinguish base-type– from function-type values: The multiplication count annotation/interpretation should hence depend on the expression type.

The multiplication-counting interpreter in the higher-order case may then look as follows:

```

module N = struct
  type  $\alpha$  exp =  $\alpha$  val_count * int
  and  $\alpha$  val_count =
    | B : float val_count
    | F : ( $\alpha$  val_count  $\rightarrow$   $\beta$  exp)  $\rightarrow$  ( $\alpha$  $\rightarrow$  $\beta$ ) val_count

  let float : float  $\rightarrow$  float exp = fun x  $\rightarrow$  (B, 0)
  let mul ((B,xc):float exp) ((B,yc):float exp) =
    (B, xc+yc+1)
  let lam (f:  $\alpha$  exp  $\rightarrow$   $\beta$  exp) =
    (F (fun vc  $\rightarrow$  f (vc,0)), 0)
  let ( $\cdot$ ) : ( $\alpha$  $\rightarrow$  $\beta$ ) exp  $\rightarrow$  ( $\alpha$  exp  $\rightarrow$   $\beta$  exp) =
    fun (F fv,fc) (xv,xc)  $\rightarrow$ 
      let (bv,bc) = fv xv in (bv, fc+xc+bc)
  end

```

Each expression is interpreted as a pair of counters: one is the count of multiplications in the expression itself; the other, α val_count, is the count of the latent multiplications that may be present in the value of the expression. The latter depends on type: base-type values have no latent computations but function-type do. Furthermore, the latent count of functions depends on their argument.

The interpreter N has two problems, seen when evaluating the earlier ex51, producing

```
(F <fun>, 0) : (float  $\rightarrow$  float) exp
```

First, no "rapp" is printed out: the sign that N disobeys the one-shot restriction. Second, the result is uninformative. As ex51 denotes a DSL function, all multiplications are latent –

performed only when the function is applied – and, hence, cannot be in general estimated statically. Although our DSL has no explicit operation for case analysis, implicit branching on the function argument is possible:

```
let exbr = lam (fun t → t · sqr · (lam (fun x → x)))
let exbr1 = exbr · (lam (fun x → lam (fun y → x · float 1.)))
let exbr2 = exbr · (lam (fun x → lam (fun y → y · float 1.)))
```

The DSL function `exbr` will do either one or zero multiplications, depending on the argument it is applied to.

The second problem is the flip side of N’s strong point: it counts exactly. Although the precise counting is attractive, in the worst case, obtaining the exact count may take about the same time and resources as just running the program in the first place – and may fail/diverge, if the program does.

As with many things, the solution is to relax: accept the possibility the counting may be too difficult, and give up. Although defeatists, the approach is surprisingly quite more useful than the exact counting: not only it is fast and always succeeding; it may report exact counts in the cases the exact approach cannot.

Here is one “relaxed” abstract interpreter, called NA. As in the exact-counting N, an expression is interpreted as a pair, of the count of expression’s multiplications and the count of multiplications latent in the expression’s value.

```
type  $\alpha$  exp =  $\alpha$  val_count * int
and  $\alpha$  val_count =
  | B : float val_count
  | F :  $\beta$  exp → ( $\alpha$ → $\beta$ ) val_count
  | Arg :  $\alpha$  val_count
  | Top :  $\alpha$  val_count
```

The latter is represented by a α val_count variant: B for base-type values (no latent counts), and F v for function-type values. Here, v is the generally approximate count of the function body – what can be statically determined, without knowing the function’s argument. There is also Top: the interpretation (Top,*n*) means that *n* is a lower bound for the expression’s multiplication count. The counts hence may be approximate. The degree of approximation is determined by the partial order:

$$(v_1, n_1) \sqsubseteq (\text{Top}, n_2) \quad \text{provided } n_2 \leq n_1$$

$$(F v_1, n) \sqsubseteq (F v_2, n) \quad \text{provided } v_1 \sqsubseteq v_2$$

Thus (Top,0) is the maximal, the least informative interpretation. The α val_count also has Arg variant, for a function argument. It is resolved to either B or F variant when the type of the argument becomes known, from the context.

The multiplication counting for the first-order DSL fragment is straightforward. The only difference from N is propagating Top:

```
let mul : float exp → float exp → float exp =
  fun (xv,xc) (yv,yc) →
    let rv = match (xv,yv) with
```

```
  | (Top,_) | (_,Top) → Top
  | _ → B
  in (rv,xc+yc+1)
```

The higher-order fragment is no more complex:

```
let lam : ( $\alpha$  exp→ $\beta$  exp) → ( $\alpha$ → $\beta$ ) exp = fun f →
  (F (f (Arg,0)), 0)
let (·) : ( $\alpha$ → $\beta$ ) exp → ( $\alpha$  exp→ $\beta$  exp) = fun (fv,fc) (xv,xc) →
  let c = fc+xc in
  match fv with
  | Arg | Top → (Top, c)
  | F (bv,bc) → (bv,c+bc)
```

The NA implementation is clearly one-shot.

Evaluating `ex51` in this implementation prints “rapp” twice and returns

```
(F (B, 1), 0) : (float → float) exp
```

which is a significantly better than the result of the exact interpreter N. Although NA generally counts only approximately, in this example it produced the exact result (no Tops): `ex51` denotes a DSL function that, when applied to any argument, does one multiplication. NA also returns the exact count for `smul`: two multiplications, when applied to two arguments. For the `ex2` example, §4, that uses a second-order function, the result is (F (Top, 1), 0): it is a function that does at least one multiplication. With second and higher-order functions, the counting is rarely exact; but at least we obtained a useful lower-bound. Finally, the branching example `exbr` yields (F (Top, 0), 0), which is to be expected: `exbr` truly denotes a function that may do any number of multiplications depending on the argument.

4.3 Pairing in the Higher-Order Case

The product `Product(L)(R)` – the higher-order version of `BProduct(L)(R)` for the `mulproc` case of §3.2 – differs from `BProduct(L)(R)` only in more general types (see the accompanying code).

As an example, interpreting `ex51` with `Product(OC)(NA)` (both OC and NA are one-shot) produces the pair

```
(.<fun x_1 →
  (fun x_2 → (fun x_3 → x_3 *. x_2) (x_1 +. x_2))
  (x_1 +. 1.)>.,
  (F (B, 1), 0))
```

of the generated OCaml code and its multiplication count.

4.4 Representing Binding

Our function generators `proc1` and `lam` used metalanguage (function-) bound variables to represent DSL bound variables – the approach that came to be called higher-order abstract syntax (HOAS) and that goes as far back as to Church [5]. Its attraction is letting DSL programmers name variables as they think fit, representing alpha-equivalent classes of

terms (so-called hygiene), and providing the substitution “for free”.

HOAS is not without problems, enumerated in Sheard’s comprehensive survey [44, §13]: junk terms (metalanguage functions that analyze the representation of object-level bound-variables), loss of expressivity (e.g., difficulty in pretty-printing HOAS terms), and latent effects.

We have seen the latent effect problem already in §2.2, as the failure of the R interpreter to count – analyzed in detail in §2.3, as the ‘wrong order’ of generating body of the function and its header (binder). The following three ho (see §4) terms illustrate the latent effect problem more forcefully.

```
let w1 = lam (fun x → while true do () done)
let w2 = lam (fun x → failwith "failed")
let w3 = lam (fun x → lam (fun y →
  if Random.bool () then x else y))
```

All three have the type $(\text{int} \rightarrow \text{int} \rightarrow \text{int}) \text{ exp}$. In the naive meta-circular R interpreter, lam is the identity function and all three terms immediately evaluate to a $(\text{int} \rightarrow \text{int} \rightarrow \text{int}) \text{ exp}$ value (which is a $(\text{int} \rightarrow \text{int} \rightarrow \text{int}) \text{ OCaml function}$). Since values of an $\alpha \text{ exp}$ type are meant to represent DSL terms, the three w1 terms must denote some DSL functions. However, there are no DSL functions that have the behavior matching these terms: our ho DSL has no effects.

As Sheard noted “HOAS delays non-termination and other effects. Computational effects of the meta-language are introduced into the purely syntactic representation of the object language. Even worse, the effects are only introduced when the object-term is observed. If a term is observed multiple times, it causes the effects to be introduced multiple times.” Sheard therefore advocated a new binding mechanism, other than lambda-abstraction, that does evaluate under binder – such as the one informally proposed by Miller [42].

In the tagless-final framework of the present paper, the problem has a much simpler solution, requiring no new binding mechanisms: we may just require lam to apply its argument, exactly once. This is the one-shot restriction. The effect is the same as evaluating under binder.

With the one-shot restriction on lam, the w1 terms above are no longer problematic. The terms w1 and w2 do not evaluate to any value – hence do not represent any DSL terms. The term w3 does evaluate to a $(\text{int} \rightarrow \text{int} \rightarrow \text{int}) \text{ exp}$ value, which represents either K or a flipped K (i.e., SK) combinator. Although the DSL generation is non-deterministic, the result is a bona fide DSL function.

We must stress that trying to solve latent effects problem by prohibiting any effects (e.g., using a “pure” metalanguage) is unsatisfactory. Effects are useful: if a code generator is not allowed to give up, consult external resources or ask a programmer for a hint, it constricts the programs one may generate. Non-deterministic program generation, as in w3 is also useful, e.g., in test case generation.

5 Conclusions and Future Challenges

The type and the very form, in the OC implementation, of the function code generators proc1 and lam betrays η -expansion $f = \lambda x. f x$ – called “the trick” in partial evaluation [12]. We have presented generically building direct product, or pairing, of the tricks, in any effectful metalanguage – provided the tricks obey the so-called one-shot restriction. In genuine code generation, function code generators are already one-shot. We have shown how to build such one-shot η -functions even for meta-circular and abstract interpreters. The trick may be regarded as an object-level binder, in HOAS. The one-shot restriction eliminates the last, hitherto unsolved problem with HOAS: latent effects. (The other problems, including lack of expressivity and junk terms, are solved by the tagless-final style [4]).

The take-home message is hence: *Tricks may be paired if they are one-shot.*

In the spirit of Danvy challenges, we finish with a few challenges of our own:

- Add the conditional expression to our DSLs. Abstract interpretation becomes more interesting.
- Make the approximate abstract interpreter NA in §4.2 build the code of a function to compute the exact count in cases it cannot be estimated statically. Since the code can be printed, we may at least get an idea what determines the exact count. The code representation is left to the reader. (Hint: it may be a formula to pass to an SMT solver, for example.)
- Consider generating quite more optimal power, with fewer or as few multiplications as possible. A promising approach:

```
let (n1,n2) = split n in mul (power n1 x) (power n2 x)
were split non-deterministically splits an integer as a
sum of two non-negative integers. To implement the
needed sharing, the method §2.5 (virtual let-binding
annotation) may be useful. The obvious application
is raising to a very large power, as used in modern
cryptographic algorithms, for example.
```

- Consider giving lam an additional argument: some representation of the type, or annotation, for the argument of the generated function. For example:

```
type  $\alpha \text{ ev}$ 
lam :  $\alpha \text{ ev} * (\alpha \text{ exp} \rightarrow \beta \text{ exp}) \rightarrow (\alpha \rightarrow \beta) \text{ exp}$ 
```

Acknowledgments

I owe much gratitude to Olivier Danvy for his many thought-provoking challenges. Numerous comments by the anonymous reviewers helped improve the presentation and are gratefully acknowledged. This work was partially supported by JSPS KAKENHI Grants Numbers 21K11821, 22H03563 and 23K24819.

References

- [1] Vincent Balat and Olivier Danvy. 2002. Memoization in Type-Directed Partial Evaluation. In *GPCE (Pittsburgh, PA) (Lecture Notes in Computer Science, 2487)*. 78–92.
- [2] Anders Bondorf. 1992. Improving Binding Times Without Explicit CPS-Conversion. In *Lisp & Functional Programming (San Francisco, CA)*. 1–10. doi:10.1145/141471.141483
- [3] Cristiano Calcagno, Walid Taha, Liwen Huang, and Xavier Leroy. 2003. Implementing Multi-stage Languages Using ASTs, Gensym, and Reflection. In *GPCE (Erfurt, Germany) (Lecture Notes in Computer Science, 2830)*. 57–76. doi:10.1007/978-3-540-39815-8_4
- [4] Jacques Carette, Oleg Kiselyov, and Chung-chieh Shan. 2009. Finally Tagless, Partially Evaluated: Tagless Staged Interpreters for Simpler Typed Languages. *Journal of Functional Programming* 19, 5 (2009), 509–543.
- [5] Alonzo Church. 1940. A Formulation of the Simple Theory of Types. *Journal of Symbolic Logic* 5, 2 (June 1940), 56–68.
- [6] Charles Consel and Olivier Danvy. 1993. Tutorial Notes on Partial Evaluation. 493–501.
- [7] Olivier Danvy. 1996. Type-Directed Partial Evaluation. In *POPL (St. Petersburg Beach, FL)*. 242–257.
- [8] Olivier Danvy. 1998. Functional Unparsing. *Journal of Functional Programming* 8, 6 (Nov. 1998), 621–625.
- [9] Olivier Danvy. 2006. An Analytical Approach to Programs as Data Objects. Doctor Scientiarum dissertation.
- [10] Olivier Danvy and Andrzej Filinski. 1989. *A Functional Abstraction of Typed Contexts*. Technical Report 89/12. DIKU, University of Copenhagen, Denmark.
- [11] Olivier Danvy and Andrzej Filinski. 1990. Abstracting Control. In *Lisp & Functional Programming (Nice, France)*. 151–160.
- [12] Olivier Danvy, Karoline Malmkjær, and Jens Palsberg. 1996. Eta-Expansion Does The Trick. *ACM Transactions on Programming Languages and Systems* 18, 6 (1996), 730–751.
- [13] Jason Eckhardt, Roumen Kaiabachev, Emir Pasalic, Kedar N. Swadi, and Walid Taha. 2007. Implicitly Heterogeneous Multi-Stage Programming. *New Generation Computing* 25, 3 (2007), 305–336. doi:10.1007/s00354-007-0020-x
- [14] A.P. Ershov. 1977. A Theoretical Principle of System Programming. *Doklady AN SSSR (Soviet Mathematics Doklady)* 18, 2 (1977), 312–315.
- [15] Andrei Petrovich Ershov. 1958. On Programming Arithmetic Operators. *Doklady Akademii Nauk* 118, 3 (1958), 427–430.
- [16] Andrei Petrovich Ershov. 1958. On Programming of Arithmetic Operations. *Comm. ACM* 1, 8 (Aug. 1958), 3–6.
- [17] Andrei P. Ershov. 1977. On the Partial Computation Principle. *IPL: Information Processing Letters* 6, 2 (1977), 38–41.
- [18] Joseph A. Goguen. 1988. *Higher Order Functions Considered Unnecessary for Higher Order Programming*. Technical Report SRI-CSL-88-1. Computer Science Laboratory, SRI International.
- [19] J. A. Goguen, J. W. Thatcher, E. G. Wagner, and J. B. Wright. 1977. Initial Algebra Semantics and Continuous Algebras. *Journal of the ACM* 24, 1 (Jan. 1977), 68–95.
- [20] M. Gordon, R. Milner, L. Morris, M. Newey, and 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, 119–130. <http://www-public.tem-tsp.eu/~gibson/Teaching/CSC4504/ReadingMaterial/GordonMMNW78.pdf>
- [21] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. 1993. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, Englewood Cliffs, NJ. <http://www.itu.dk/people/sestoft/pebook/pebook.html>
- [22] Neil D. Jones and Flemming Nielson. 1994. Abstract Interpretation: a Semantics-Based Tool for Program Analysis. In *Handbook of Logic in Computer Science*. Oxford University Press, 527–629.
- [23] Yuki Yoshi Kameyama, Oleg Kiselyov, and Chung-chieh Shan. 2008. Closing the Stage: From Staged Code to Typed Closures. In *PEPM (San Francisco, CA)*. ACM Press, New York, 147–157. doi:10.1145/1328408.1328430
- [24] Yuki Yoshi Kameyama, Oleg Kiselyov, and Chung-chieh Shan. 2011. Shifting the Stage: Staging with Delimited Control. *Journal of Functional Programming* 21, 6 (2011), 617–662. doi:10.1017/S0956796811000256
- [25] Yuki Yoshi Kameyama, Oleg Kiselyov, and Chung-chieh Shan. 2015. Combinators for impure yet hygienic code generation. *Science of Computer Programming* 112 (part 2) (15 Nov. 2015), 120–144. doi:10.1016/j.scico.2015.08.007
- [26] Samuel Kamin. 1996. Standard ML as a Meta-Programming Language. <http://loome.cs.uiuc.edu/pubs.html>.
- [27] Sam Kamin, Miranda Callahan, and Lars Clausen. 2000. Lightweight and Generative Components I: Source-Level Components. In *Proc. GCSE'99 (Lecture Notes in Computer Science, Vol. 1799)*. Springer, 49–62.
- [28] Oleg Kiselyov. 2011. Implementing Explicit and Finding Implicit Sharing in Embedded DSLs. In *DSL*. 210–225. doi:10.4204/EPTCS.66.11
- [29] Oleg Kiselyov. 2012. Typed Tagless Final Interpreters. In *Proceedings of the 2010 International Spring School Conference on Generic and Indexed Programming*. Springer-Verlag, Berlin, Heidelberg, 130–174. doi:10.1007/978-3-642-32202-0_3
- [30] Oleg Kiselyov. 2014. The Design and Implementation of BER MetaOCaml - System Description. In *FLOPS (Kanazawa, Japan) (Lecture Notes in Computer Science, 8475)*. Springer, 86–102. doi:10.1007/978-3-319-07151-0_6
- [31] Oleg Kiselyov. 2017. Generating Code with Polymorphic let: A Ballad of Value Restriction, Copying and Sharing. *EPTCS* 241 (2017), 1–22. doi:10.4204/EPTCS.241.1
- [32] Oleg Kiselyov. 2017. Higher-order Programming is an Effect. HOPE 2017 at ICFP 2017. <https://okmij.org/ftp/Computation/having-effect.html>
- [33] Oleg Kiselyov. 2019. Effects Without Monads: Non-determinism – Back to the Meta Language. *Electronic proceedings in theoretical computer science* 294 (2019), 15–40. doi:10.4204/EPTCS.294.2
- [34] Oleg Kiselyov. 2023. Free Variable as Effect, in Practice. doi:10.48550/arXiv.2312.16446 HOPE Workshop 2023.
- [35] Oleg Kiselyov. 2023. Generating C: Heterogeneous Metaprogramming (System Description). *Sci. Comput. Program.* 231 (2023), 103015. doi:10.1016/J.SCICO.2023.103015
- [36] Oleg Kiselyov. 2024. MetaOCaml: Ten Years Later - System Description. In *Functional and Logic Programming - 17th International Symposium, FLOPS 2024, May 15-17, 2024, Proceedings (Kumamoto, Japan) (Lecture Notes in Computer Science, Vol. 14659)*, Jeremy Gibbons and Dale Miller (Eds.). Springer, 219–236. doi:10.1007/978-981-97-2300-3_12
- [37] Oleg Kiselyov. 2025. BER MetaOCaml N153. <https://okmij.org/ftp/ML/MetaOCaml.html>.
- [38] Oleg Kiselyov and Keigo Imai. 2020. Session Types without Sophistry. In *Functional and Logic Programming (Lecture Notes in Computer Science, Vol. 12073)*. Springer International Publishing, 66–87. doi:10.1007/978-3-030-59025-3_5
- [39] Oleg Kiselyov, Yuki Yoshi Kameyama, and Yuto Sudo. 2016. Refined Environment Classifiers - Type- and Scope-Safe Code Generation with Mutable Cells. In *Proceedings of the Programming Languages and Systems - 14th Asian Symposium, APLAS 2016, November 21-23, 2016 (Hanoi, Vietnam) (Lecture Notes in Computer Science, Vol. 10017)*, Atsushi Igarashi (Ed.). Springer-Verlag, 271–291. doi:10.1007/978-3-319-47958-3_15
- [40] Oleg Kiselyov, Chung-chieh Shan, and Amr Sabry. 2006. Delimited Dynamic Binding. In *ICFP (Portland, OR)*. 26–37.
- [41] Julia L. Lawall and Olivier Danvy. 1994. Continuation-Based Partial Evaluation. In *Lisp & Functional Programming (Orlando, FL)*. 227–238. doi:10.1145/182409.182483

- [42] Dale Miller. 1990. An Extension to ML to Handle Bound Variables in Data Structures: Preliminary Report. In *Informal Proceedings of the Logical Frameworks BRA Workshop*. Available as UPenn CIS technical report MS-CIS-90-59.
- [43] Frank Pfenning. 2008. Church and Curry: Combining Intrinsic and Extrinsic Typing. In *Festschrift in honour of Peter B. Andrews on His 70th Birthday*, Christoph Benzmüller, Chad Brown, Jörg Siekmann, and Rick Statman (Eds.). IFCoLog. <http://www.cs.cmu.edu/~fp/papers/andrews08.pdf>
- [44] Tim Sheard. 2001. Accomplishments and Research Challenges in Meta-programming. In *Proceedings of SAIG 2001: 2nd International Workshop on Semantics, Applications, and Implementation of Program Generation* (Florence, Italy) (*Lecture Notes in Computer Science*, 2196), Walid Taha (Ed.). Springer-Verlag, Berlin, 2–44.
- [45] Kedar Swadi, Walid Taha, Oleg Kiselyov, and Emir Pašalić. 2006. A Monadic Approach for Avoiding Code Duplication When Staging Memoized Functions. In *PEPM* (Charleston, SC). 160–169.
- [46] Walid Taha. 1999. *Multi-Stage Programming: Its Theory and Applications*. Ph.D. Dissertation. Oregon Graduate Institute of Science and Technology.

Received 2025-06-11; accepted 2025-07-31