Phases in Software Architecture

Jeremy Gibbons
jeremy.gibbons@cs.ox.ac.uk
University of Oxford
Oxford, UK

Donnacha Oisín Kidney
o.kidney21@imperial.ac.uk
Imperial College London
London, UK

Tom Schrijvers
tom.schrijvers@kuleuven.be
KU Leuven
Leuven, BE

Nicolas Wu
n.wu@imperial.ac.uk
Imperial College London
London, UK

Abstract

The large-scale structure of executing a computation can often be thought of as being separated into distinct *phases*. But the most natural form in which to *specify* that computation may well have a different and conflicting structure. For example, the computation might consist of gathering data from some locations, processing it, then distributing the results back to the same locations; it may be executed in three phases—gather, process, distribute—but mostly conveniently specified orthogonally—by location. We have recently shown that this multi-phase structure can be expressed as a novel *applicative functor* (also known as an *idiom*, or *lax monoidal* functor). Here we summarize the idea from the perspective of software architecture. At the end, we speculate about applications to choreography and multi-tier architecture.

CCS Concepts: • **Software and its engineering** → **Software system structures**; **Abstraction, modeling and modularity**; **Correctness**; *Functional languages*; *Control structures*; Patterns; • **Theory of computation** → *Control primitives*; **Program reasoning**.

Keywords: traversal, applicative functor, fusion, phase separation, choreography, multi-tier

ACM Reference Format:
Jeremy Gibbons, Donnacha Oisín Kidney, Tom Schrijvers, and Nicolas Wu. 2023. Phases in Software Architecture. In *Proceedings of the 1st ACM SIGPLAN International Workshop on Functional Software Architecture (FUNARCH '23), September 8, 2023, Seattle, WA, USA.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3609025.3609479

FUNARCH '23, September 8, 2023, Seattle, WA, USA

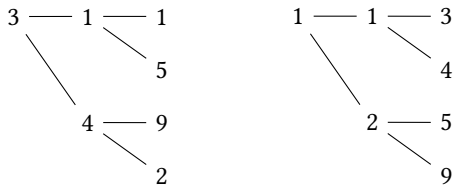
ACM ISBN 979-8-4007-0297-6/23/09.
https://doi.org/10.1145/3609025.3609479

1 Introduction

Consider the problem of sorting the elements of a tree. Given a tree datatype:

data *Tree a* = *Node a* (*Forest a*)
type *Forest a* = [*Tree a*]

and an input tree as on the left below:



the problem is to extract the elements [3, 1, 4, 1, 5, 9, 2] of the tree through an in-order traversal, sort this list to obtain [1, 1, 2, 3, 4, 5, 9], then replace the elements in the new order to get the tree on the right. The problem is admittedly artificial as stated, but it was introduced by Bird [1] as an abstraction of compilation tasks like traversing an abstract syntax tree once in order to gather declarations, resolve any forward references, then distribute the results back over the tree.

Of course, the problem can be solved directly by a program following that description: traverse the tree to get a list, sort the list, traverse the tree again to distribute the list. And indeed, operationally that may be what should happen. However, the two traversals have the same structure, and it is clumsy to have to specify this structure twice. Bird's argument was that lazy evaluation (in particular, **letrec**) allows one to express the program with a single traversal of the tree, by *fusing* the two traversals into one. One may argue about whether Bird's solution actually leads *dynamically* to a single traversal of the tree, but that argument is not our concern here: it is clear that Bird's program *statically* describes only a single traversal.

In response to Bird's paper, Pettorossi [13, 14] showed that laziness and circularity are not actually needed: higher-order lambda abstraction suffices. We have shown [7] that in fact Bird's and Pettorossi's solutions are really essentially

the same. One may view them both as specifying a multi-phase computation (gather, process, distribute) then executing it. Bird uses laziness to delay certain evaluation until it is needed; Pettorossi constructs a closure before applying it to the needed value. We explicate the multi-phase computation that is the essence of both solutions.

Concretely, we define a type *Phases m a* parametrized by a type m of effects and a return type a, representing a multi-phase computation with effect in m and returning an a. When m is an *Applicative* functor [10], so too is *Phases m*. There are functions

$$\mathit{phase} :: \mathit{Applicative}\ m \Rightarrow \mathit{Int} \to m\ a \to \mathit{Phases}\ m\ a$$
$$\mathit{runPhases} :: \mathit{Applicative}\ m \Rightarrow \mathit{Phases}\ m\ a \to m\ a$$

to inject a computation in one phase of a multi-phase computation, and to sequence the phases. Sorting a tree uses the effects *State* [a] to act on a state consisting of the list of elements:

$$\mathit{sortTree} :: \mathit{Ord}\ a \Rightarrow \mathit{Tree}\ a \to \mathit{Tree}\ a$$
$$\mathit{sortTree}\ t = \mathit{evalState}\ (\mathit{runPhases}\ (\mathit{sortTreeAux}\ t))\ [\,]$$

$$\mathit{sortTreeAux} :: \mathit{Ord}\ a \Rightarrow \mathit{Tree}\ a \to \mathit{Phases}\ (\mathit{State}\ [\,a\,])\ (\mathit{Tree}\ a)$$
$$\begin{aligned}&\mathit{sortTreeAux}\ t\\ &\quad = \mathit{phase}\ 1\ (\mathit{traverse}\ (\lambda x \to \mathit{push}\ x)\ t) \mathbin{*\!\!>}\\ &\qquad \mathit{phase}\ 2\ (\mathit{modify}\ \mathit{sort}) \mathbin{*\!\!>}\\ &\qquad \mathit{phase}\ 3\ (\mathit{traverse}\ (\lambda x \to \mathit{pop})\ t)\end{aligned}$$

(here, $\mathit{evalState} :: \mathit{State}\ s\ a \to (s \to a)$ is a standard function for the *State* monad). The traversal in phase 1 gathers elements; phase 2 sorts the list, without touching the tree; the traversal in phase 3 distributes the elements back over the tree. Crucially, the *specifications* of actions in different phases commute, even when their *executions* do not; after all, "do X now and Y later" is equivalent to "do Y later and X now". So we can permute the phases to bring the two traversals together:

$$\begin{aligned}&\mathit{sortTreeAux}\ t\\ &\quad = \mathit{phase}\ 2\ (\mathit{modify}\ \mathit{sort}) \mathbin{*\!\!>}\\ &\qquad \mathit{phase}\ 1\ (\mathit{traverse}\ (\lambda x \to \mathit{push}\ x)\ t) \mathbin{*\!\!>}\\ &\qquad \mathit{phase}\ 3\ (\mathit{traverse}\ (\lambda x \to \mathit{pop})\ t)\end{aligned}$$

and then fuse the two traversals into one:

$$\begin{aligned}&\mathit{sortTreeAux}\ t\\ &\quad = \mathit{phase}\ 2\ (\mathit{modify}\ \mathit{sort}) \mathbin{*\!\!>}\\ &\qquad \mathit{traverse}\ (\lambda x \to \mathit{phase}\ 1\ (\mathit{push}\ x) \mathbin{*\!\!>} \mathit{phase}\ 3\ \mathit{pop})\ t\end{aligned}$$

which has manifestly only a single traversal. The details are in the paper [7].

Our purpose in this short note is to raise awareness of the multi-phase construction, in the hope that it can be a useful functional software architecture technique. We can merely sketch the ideas here, but the full details are in the paper.

2 Applicative Programming with Effects

Moggi [11] and Wadler [18] famously showed that effectful programs could be written in a pure functional language using *monads*. McBride and Paterson [10] showed later that the slightly less powerful abstraction of *applicative functors* often suffice, with some advantages. We will use the latter:

$$\begin{aligned}&\textbf{class}\ \mathit{Functor}\ m \Rightarrow \mathit{Applicative}\ m\ \textbf{where}\\ &\quad \mathit{pure} :: a \to m\ a\\ &\quad (\langle * \rangle) :: m\ (a \to b) \to m\ a \to m\ b\end{aligned}$$

(subject to some laws, omitted for brevity). Thus, plain values can be lifted to pure computations, and one can sequence two computations and combine their results. From this we can derive combination by pairing:

$$(\otimes) :: \mathit{Applicative}\ m \Rightarrow m\ a \to m\ b \to m\ (a, b)$$
$$xs \otimes ys = \mathit{pure}\ (,) \langle * \rangle xs \langle * \rangle ys$$

(here, $(,) :: a \to b \to (a, b)$ is curried pairing) and biased sequencing, discarding the result from the first computation:

$$(*\rangle) :: \mathit{Applicative}\ m \Rightarrow m\ a \to m\ b \to m\ b$$
$$xs *\rangle ys = \mathit{fmap}\ \mathit{snd}\ (xs \otimes ys)$$

both of which we will use later.

Applicative traversal is "the essence of the Iterator design pattern" [6], capturing computations that iterate over a data structure, in a predetermined order, processing each element in turn and collecting effects as they go:

$$\begin{aligned}&\textbf{class}\ \mathit{Functor}\ t \Rightarrow \mathit{Traversable}\ t\ \textbf{where}\\ &\quad \mathit{traverse} :: \mathit{Applicative}\ m \Rightarrow (a \to m\ b) \to t\ a \to m\ (t\ b)\end{aligned}$$

(again, subject to some omitted laws). For example, here are left-to-right list traversal and in-order tree traversal:

$$\begin{aligned}&\textbf{instance}\ \mathit{Traversable}\ [\,]\ \textbf{where}\\ &\quad \mathit{traverse}\ f\ [\,] \qquad = \mathit{pure}\ [\,]\\ &\quad \mathit{traverse}\ f\ (x : xs) = \mathit{pure}\ (:) \langle * \rangle f\ x \langle * \rangle \mathit{traverse}\ f\ xs\end{aligned}$$

$$\begin{aligned}&\textbf{instance}\ \mathit{Traversable}\ \mathit{Tree}\ \textbf{where}\\ &\quad \mathit{traverse}\ f\ (\mathit{Node}\ x\ ts) = \mathit{pure}\ \mathit{Node} \langle * \rangle f\ x \langle * \rangle\\ &\qquad\qquad\qquad\qquad \mathit{traverse}\ (\mathit{traverse}\ f)\ ts\end{aligned}$$

Now consider the composition $\mathit{traverse}\ f\ t *\rangle \mathit{traverse}\ g\ t$ of two traversals, discarding the results of the first, and similarly a single traversal $\mathit{traverse}\ (\lambda x \to f\ x *\rangle g\ x)\ t$ with the composition of the two bodies. In general, these cannot be equal: the former performs all f-effects before any g-effects, while the latter interleaves them. If the class of effects were commutative, the interleaving would not matter; but that condition is very restrictive. Happily, commutativity of the whole class of effects is not necessary; it suffices for f-effects to commute with g-effects:

$$f\ x \otimes g\ y = \mathit{fmap}\ \mathit{twist}\ (g\ y \otimes f\ x)$$

where $\mathit{twist}\ (x, y) = (y, x)$. That is, running f then g and pairing the results is the same as running g then f and flip-pairing the results. The proof is in the paper [7].

3 Two Phases

It turns out that two-phase computation can be captured precisely by what is known as *Day convolution* [4]:

```
data Day m n a where
  Day :: (a → b → c) → m a → n b → Day m n c
```

(perhaps not surprising, given that Day convolution and applicative functors are deeply connected [15]). Thus, *Day f xs ys* represents a two-phase computation, with subcomputation *xs* happening in phase one generating effects in *m*, and *ys* in phase two generating effects in *n*, packaged up with a function *f* to combine the results from the two phases. Moreover, *Day m n* is an applicative functor when *m* and *n* both are:

```
instance (Applicative m, Applicative n) ⇒
         Applicative (Day m n) where
  pure x = Day (curry fst) (pure x) (pure ())
  Day f xs ys ‹*› Day g zs ws = Day h (xs ⊗ zs) (ys ⊗ ws)
    where h (x, z) (y, w) = (f x y) (g z w)
```

(here, *curry fst* :: $a \to () \to a$ is half of the right unit isomorphism of products). We can inject into either phase, using and discarding a trivial computation for the other phase:

```
phase1 :: (Applicative m, Applicative n) ⇒ m a → Day m n a
phase1 xs = Day (curry fst) xs (pure ())

phase2 :: (Applicative m, Applicative n) ⇒ n a → Day m n a
phase2 ys = Day (curry snd) (pure ()) ys
```

When the two classes of effects coincide, we can combine the two phases, running one after the other and post-processing the results:

```
runDay :: Applicative m ⇒ Day m m a → m a
runDay (Day f xs ys) = pure f ‹*› xs ‹*› ys
```

Crucially for us, computations in different phases commute:

```
phase1 xs ⊗ phase2 ys = fmap twist (phase2 ys ⊗ phase1 xs)
```

For example, we can send a two-part greeting in separate phases:

```
⋙ runDay (phase1 (putStr "Hello ") ‹*›
          phase2 (putStr "World"))
```

Hello World

It doesn't matter if we specify those two phases in the opposite order:

```
⋙ runDay (phase2 (putStr "World") ‹*›
          phase1 (putStr "Hello "))
```

Hello World

We can even interleave the specification of fragments from different phases:

```
⋙ runDay (phase1 (putStr "Hel") ‹*›
          phase2 (putStr "World") ‹*›
          phase1 (putStr "lo "))
```

Hello World

4 Multiple Phases

We now generalize from two-phase computations to multiple (zero or more) phases [5]:

```
data Phases m a where
  Pure :: a → Phases m a
  Link :: (a → b → c) → m a → Phases m b → Phases m c
```

Here, *Pure* produces a chain with no effectful phases, and *Link* adds one more effectful phase to the chain. It is essentially a homogeneous iteration of Day convolution (*Link* constructs the Day convolution of *f* with *Phases f*), just as lists are essentially a homogeneous iteration of pairing (with cons pairing a list head with a tail). There is a single initial value as the base case; each additional link in the chain adds a combining function and a collection of values; and the types are all compatible "in the obvious way".

We implement an *Applicative* instance that *zips* together chains: composing *xs* and *ys* should mean "in phase 1, execute phase 1 of *xs* and then phase 1 of *ys*; in phase 2, execute phase 2 of *xs* then phase 2 of *ys*" and so on. To implement this, we need the underlying effect *m* itself to be an *Applicative* and not just a *Functor*:

```
instance Applicative m ⇒ Applicative (Phases m) where
  pure x          = Pure x
  Pure f ‹*› xs   = fmap f xs
  fs ‹*› Pure x   = fmap (λf → f x) fs
  Link f xs ys ‹*› Link g zs ws = Link h (xs ⊗ zs) (ys ⊗ ws)
    where h (x, z) (y, w) = (f x y) (g z w)
```

(Note that the same *Phases m* datatype is also the carrier of the *free applicative* induced by functor *m* [2]. Informally, this is defined in such a way that composition concatenates chains rather than zipping them, assuming only the weaker *Functor* condition on *m*.)

We can inject into any phase:

```
now :: Applicative m ⇒ m a → Phases m a
now xs = Link (curry fst) xs (Pure ())

later :: Applicative m ⇒ Phases m a → Phases m a
later xs = Link (curry snd) (pure ()) xs

phase :: Applicative m ⇒ Int → m a → Phases m a
phase 1 = now
phase i = later ∘ phase (i − 1)
```

and sequence together the phases:

```
runPhases :: Applicative m ⇒ Phases m a → m a
runPhases (Pure x)          = pure x
runPhases (Link f xs ys) = pure f ‹*› xs ‹*› runPhases ys
```

And again, computations in different phases commute:

```
phase i xs ⊗ phase j ys = fmap twist (phase j ys ⊗ phase i xs)
```

provided $i \neq j$.

5 Examples

We have seen the outline of the tree-sorting example already. The remaining details are that *push x* and *pop* manipulate the list stored in the state:

```
push :: a → State [a] ()
push x = modify (x:)

pop :: State [a] a
pop = do {x : xs ← get; put xs; return x}
```

(here, *get*, *put*, and *modify* are more standard functions for the *State* monad).

5.1 Repmin

A related example, also from Bird's paper [1] and also addressed by Pettorossi [13], is to replace every element of a tree with the minimum element in the tree. Bird solves the problem with a circular lazy program, computing both the minimum and the new tree in a single pass; Pettorossi uses a higher-order abstraction, computing the minimum and a *function* from a value to a constant tree, then applies the latter to the former. We [7] show a single traversal yielding a two-phase computation

```
repminAux :: Tree Int → Day WInt RInt (Tree Int)
repminAux
  = traverse (λx → phase1 (tellMin x) ∗› phase2 askMin)
```

where *WInt* is the *Writer* monad on minimizing *Int*s:

```
type WInt = Writer (Min Int)

tellMin :: Int → WInt ()
tellMin x = tell (Min x)
```

and *RInt* is the *Reader* monad on the same type:

```
type RInt = Reader (Min Int)

askMin :: RInt Int
askMin = fmap getMin ask
```

This core is common to Bird's and Pettorossi's solutions; their difference is in how to extract the two phases. (Because two different classes of effect are involved, *runDay* isn't applicable.) Bird unwraps the writer and reader computation in parallel:

```
parWR :: Day (Writer s) (Reader s) a → a
parWR (Day f xs ys)
  = let ((x, s), y) = (runWriter xs, runReader ys s)
    in f x y

repmin_RSB :: Tree Int → Tree Int
repmin_RSB t = parWR (repminAux t)
```

Note that *parWR* is circular, with *s* appearing on both sides of the local declaration, so the **let** must have **letrec** semantics. In contrast, Pettorossi unwraps the writer and reader computations sequentially:

```
seqWR :: Day (Writer s) (Reader s) a → a
seqWR (Day f xs ys) = let (x, s) = runWriter xs
                          y      = runReader ys s
                      in f x y

repmin_ADP :: Tree Int → Tree Int
repmin_ADP t = seqWR (repminAux t)
```

Now there is no circularity, and a plain non-recursive **let** suffices. Moreover, in a lazy language, clearly *parWR* and *seqWR* are equal, and so too therefore are $repmin_{\text{RSB}}$ and $repmin_{\text{ADP}}$.

5.2 Breadth-first Traversal

A third example concerns breadth-first traversal of trees. Depth-first traversal is straightforward, because it is obviously compositional with respect to the tree structure:

```
dft :: Applicative m ⇒ (a → m b) → Tree a → m (Tree b)
dft f (Node x ts) = pure Node ‹∗› f x ‹∗› traverse (dft f) ts
```

(this definition is equivalent to the *Traversable* instance shown in Section 2). To obtain breadth-first traversal, it suffices to schedule the visits to different levels in different phases, using *now* and *later*:

```
bft′ :: Applicative m ⇒
        (a → m b) → Tree a → Phases m (Tree b)
bft′ f (Node x ts)
  = pure Node ‹∗› now (f x) ‹∗› later (traverse (bft′ f) ts)
```

The root label *x* is processed 'now'; a multi-phase computation is constructed for each child in *ts*, zipped together by levels using *traverse* on lists, then postponed until one phase 'later'. To obtain something of the right type for *Traversable*, we just have to flatten the phases:

```
bft :: Applicative m ⇒ (a → m b) → Tree a → m (Tree b)
bft f = runPhases ∘ bft′ f
```

In particular, we can relabel a tree in breadth-first order, needing neither queues [12] nor cyclicity and laziness [9]:

```
bfl :: Tree a → [b] → Tree b
bfl t xs = evalState (bft (λx → pop) t) xs
```

6 Discussion

Nearly fifty years ago, Jackson's influential book [8] argued that programs are clearest when they follow the structure of the data they consume and produce; and consequently, that the knottiest problems in software architecture are when these structures clash. A rather canonical instance of that phenomenon is provided by breadth-first traversal, which very much goes "against the grain" of the tree structure. We expect that there are other thorny problems in software architecture that may be susceptible to this multi-phase approach.

The tree-sorting and repmin examples are different in kind to breadth-first traversal. Although we used *Phases* for tree-sorting, it is similar to repmin in the sense that the number of phases is statically determined (three for sorting, two for repmin), whereas for breadth-first traversal it is only dynamically known (determined by the depth of the tree). We likewise expect that this fixed collection of execution phases, naturally specified in a different order than they are executed, is a recurring pattern in software architecture.

In particular, we conjecture that process *choreography* is one such example. In choreographic programming, a single global program specifies a distributed system; *end-point projection* translates this to separate local programs for each node of the system. Recent work [16] has demonstrated that this translation can be captured in terms of free monads. Perhaps it can also fruitfully be expressed using *Phases*: all the applications we have described here involve phases indexed by natural numbers, which have an inherent ordering, but there seems to be no reason not to allow a computation to be split into fragments indexed instead by *location*. Similarly, we conjecture that a multi-tier application architecture [3] can be modelled as a *Phases* computation indexed by tier (for example, presentation layer, logic layer, data layer).

Our *Phases* type is homogeneous; one might wonder (and indeed, one reviewer asked) about a heterogeneous generalization. Formally, that would be no more general: the composition (or even the Day convolution) of distinct *Applicative*s is again *Applicative*, so any heterogeneous application can always be upcast to a homogeneous one. We can already get a glimpse of that in the repmin example: the 'min' phase uses only the *Writer* effect, and the 'rep' phase only the *Reader* effect. Since there are only two static phases in this example, we could get away with Day convolution, which can be heterogeneous. If we had instead a dynamic multi-phase computation, with some phases using *Reader* and not *Writer*, some *Writer* and not *Reader*, they could still all be expressed homogeneously in the combined effect *Day* (*Writer s*) (*Reader s*).

As a final thought: in retrospect, maybe "staging" wasn't the best choice of term to use in the original paper [7]. That term has a specific technical meaning in program generation [17]: an earlier phase generates *code* that is not analysed or executed until a later phase. Our meaning here is less specific: an earlier phase generates some computation (which could be an actual function, but in our approach is a data structure that represents a function) which is executed in a later phase; but all the analysis happens up front.

Acknowledgments

This extended abstract is a summary of previously published work [7]; no new results are presented here, only some very preliminary thoughts about the relevance to software architecture. The *Phases* construction and its applicative instance, and the corresponding definition of breadth-first traversal, are due originally to Easterly [5].

References

[1] Richard S. Bird. 1984. Using Circular Programs to Eliminate Multiple Traversals of Data. *Acta Informatica* 21 (1984), 239–250. https://doi.org/10.1007/BF00264249

[2] Paolo Capriotti and Ambrus Kaposi. 2014. Free Applicative Functors. In *Mathematically Structured Functional Programming (EPTCS, Vol. 153)*. 2–30. https://doi.org/10.4204/EPTCS.153.2

[3] Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. 2006. Links: Web Programming Without Tiers. In *Formal Methods for Components and Objects (LNCS, Vol. 4709)*. Springer, 266–296. https://doi.org/10.1007/978-3-540-74792-5_12

[4] Brian Day. 1970. On Closed Categories of Functors. In *Reports of the Midwest Category Seminar IV (Lecture Notes in Mathematics, Vol. 137)*. Springer-Verlag, 1–38. https://doi.org/10.1007/BFb0060438

[5] Noah Easterly. 2019. Functions and Newtype Wrappers for Traversing Trees: rampion/tree-traversals. https://github.com/rampion/tree-traversals.

[6] Jeremy Gibbons and Bruno César dos Santos Oliveira. 2009. The Essence of the Iterator Pattern. *JFP* 19, 3,4 (2009), 377–402. https://doi.org/10.1017/S0956796809007291

[7] Jeremy Gibbons, Donnacha Oisín Kidney, Tom Schrijvers, and Nicolas Wu. 2022. Breadth-First Traversal Via Staging. In *Mathematics of Program Construction (LNCS, Vol. 13544)*. Springer, 1–33. https://doi.org/10.1007/978-3-031-16912-0_1

[8] Michael A. Jackson. 1975. *Principles of Program Design*. Academic Press.

[9] Geraint Jones and Jeremy Gibbons. 1993. *Linear-time Breadth-first Tree Algorithms: An Exercise in the Arithmetic of Folds and Zips*. Working Paper 705 WIN-2. IFIP WG2.1. http://www.cs.ox.ac.uk/publications/publication2363-abstract.html

[10] Conor McBride and Ross Paterson. 2008. Applicative Programming with Effects. *JFP* 18, 1 (2008), 1–13. https://doi.org/10.1017/S0956796807006326

[11] Eugenio Moggi. 1991. Notions of Computation and Monads. *Information and Computation* 93, 1 (1991), 55–92. https://doi.org/10.1016/0890-5401(91)90052-4

[12] Chris Okasaki. 2000. Breadth-First Numbering: Lessons from a Small Exercise in Algorithm Design. In *International Conference on Functional Programming*. ACM, 131–136. https://doi.org/10.1145/351240.351253

[13] Alberto Pettorossi and Andrzej Skowron. 1987. Higher Order Generalization in Program Derivation. In *Theory and Practice of Software Development (LNCS, Vol. 250)*. Springer, 182–196. https://doi.org/10.1007/BFb0014981

[14] Alberto Pettorossi and Andrzej Skowron. 1989. The Lambda Abstraction Strategy for Program Derivation. *Fundamenta Informaticae* XII (1989), 541–562. https://doi.org/10.3233/FI-1989-12407

[15] Exequiel Rivas and Mauro Jaskelioff. 2017. Notions of Computation as Monoids. *JFP* 27 (2017), e21. https://doi.org/10.1017/S0956796817000132

[16] Gan Shen, Shun Kashiwa, and Lindsey Kuper. 2023. HasChor: Functional Choreographic Programming for All (Functional Pearl). arXiv:2303.00924 [cs.PL]

[17] Walid Taha. 2003. A Gentle Introduction to Multi-stage Programming. In *Domain-Specific Program Generation (LNCS, Vol. 3016)*. Springer, 30–50. https://doi.org/10.1007/978-3-540-25935-0_3

[18] Philip Wadler. 1992. Monads for Functional Programming. In *Program Design Calculi: Proceedings of the Marktoberdorf Summer School (NATO ASI Series F, Vol. 118)*. Springer, 233–264. https://doi.org/10.1007/978-3-662-02880-3_8

Received 2023-06-01; accepted 2023-06-28