Composing Effects into Tasks and Workflows

Yves Parès
Tweag I/O
Paris, France
yves.pares@gmail.com

Jean-Philippe Bernardy
Department of Philosophy,
Linguistics and and Theory of Science
University of Gothenburg, Sweden
Gothenburg, Sweden
jean-philippe.bernardy@gu.se

Richard A. Eisenberg
Tweag I/O
Cambridge, UK
Bryn Mawr College
Bryn Mawr, PA, USA
rae@richarde.dev

Abstract

Data science applications tend to be built by composing *tasks*: discrete manipulations of data. These tasks are arranged in directed acyclic graphs, and many frameworks exist within the data science community supporting such a structure, which is called a *workflow*. In realistic applications, we want to be able to both *analyze* a workflow in the absence of data, and to *execute* the workflow with data.

This paper combines effect handlers with arrow-like structures to abstract out data science tasks. This combination of techniques enables a modular design of workflows. Additionally, these workflows can both be analyzed prior to running (e.g., to provide early failure) and run conveniently. Our work is directly motivated by real-world scenarios, and we believe that our approach is applicable to new data science and machine learning applications and frameworks.

Keywords: Haskell, effect handlers, arrows

1 Introduction

Many data science or machine learning applications and architectures are written following the same pattern: a succession of computation *tasks*, each of them receiving a set of inputs and computing a set of outputs. However, tasks are not necessarily arranged linearly. Rather, data flows in a directed and (often) acyclic graph (DAG) of tasks. Several existing frameworks enforce this DAG structure, where the graph appears either at the level of individual mathematical computations (as in Tensorflow [Abadi et al. 2016] or Torch [Collobert et al. 2002]), at the level of processing blocks of lines from a dataset (Apache Spark [Zaharia et al. 2010], Storm[1], or Beam[2]), or at the level of orchestrating several coarse-granularity tasks (like training or classification) and/or even several programming languages and environments (Spotify's Luigi [Erdmann et al. 2017] or AirBnB's Airflow [Kotliar et al. 2019]). At any of these levels, we are likely to encounter computations chained with no conditional branching, therefore exhibiting a *fully statically manifest* DAG structure. This means that the structure of tasks is independent of the data input.

Workflows must also be able to execute *effects*: these effects might be reading input parameters, logging actions taken, or pulling data across a network. The notion of effectful computations that can be chained and analyzed before execution has existed for some time. The Haskell community is aware of several structures that allow such chaining and analysis, notably *categories* and *arrows*, as originally proposed by Hughes [2000] but studied more in subsequent work [Heunen and Jacobs 2006; Jacobs et al. 2009; Paterson 2001; Rivas and Jaskelioff 2014]; and *applicative functors* [McBride and Paterson 2007], recently extended to support conditional branching by Mokhov et al. [2019].

Despite matching the requirements for modeling workflows, neither arrows nor other binary effects (captured in a type parameterized by both an input and an output) have, to our knowledge, been applied to meet this need. Indeed, arrows seem to have remained something of a niche concept within the functional programming community, used with functional reactive programming (FRP), as proposed by Hudak et al. [2003], but little elsewhere. One challenge with arrows, as with any effect system, is in how effects compose. While well-studied in the context of monads (e.g. [Liang et al. 1995] and its sequels), effect composition has only rarely be considered with respect to arrow-like structures [Lindley 2014]. We will present a way to implement algebraic effects and their handlers in the case of arrow-like structures in a language like Haskell.

1.1 Phases

In order to better formalize the point of computations being analysed before they run, we also introduce a novel way of analyzing the behavior of workflow programs: *phases*. Programmers in compiled languages are accustomed to thinking about two phases in which their programs live: compile-time and run-time. Type checking and optimization are typically performed at compile-time, while execution of the program logic happens at run-time. The phase separation offers several nice benefits:

- Errors in the source code can often be reported at compile-time. This is convenient, because the programmer can see these errors and act on them. Run-time errors can be a bit more baffling.

[1] http://storm.apache.org
[2] http://beam.apache.org

Haskell '20, August 27, 2020, Virtual Event, USA
2020. ACM ISBN 978-1-4503-8050-8/20/08…$15.00
https://doi.org/10.1145/3406088.3409023

- Errors reported at compile-time mean that run-time work is not wasted. It is a shame when a long-running program crashes after it has done useful work, which is then lost. Finding coding errors at compile-time is one way to ensure *early failure*.
- Some computation—such as type checking—can be done at compile-time, leading to lower run-time execution times. This is helpful because compiling is done by the programmer, while run-time execution is done by the client.

We go further by splitting run-time into two phases: *config-time* and *process-time*. *Config-time* happens right after the application is loaded but before any data is processed; data processing happens in *process-time*—which can take days. This separation has benefits parallel to the benefits of separating compile-time from run-time:

- Given that a data science expert has started the application, errors reported at config-time can be interpreted and fixed instead of occurring days into the processing phase. For example, perhaps some parameters for the computations to run are invalid, or necessary files are missing on the execution platform, or a network connection is misconfigured.
- Errors reported at config-time will stop execution *before* meaningful work is performed on the data. This may mean days of computation time saved, if an error would be otherwise reported near the end of a workflow. In a scenario where the application is started as a dry-run, errors can be reported even in the absence of any data.
- It may be possible that config-time analysis can *optimize* the workflow. The computation associated with this optimization is performed *once*, in advance, instead of concurrently with data processing.

Note that config-time, as being part of the run-time, typically has access to more information than compile-time. For example, config-time might easily access configuration files or otherwise inspect the deployment environment, which may be different from the development environment used for compiling. This access would be is impractical to do at compile-time—it would require all the effectful code to be run as metaprogramming macros and would require the machine running the workflow to rebuild the application anytime it should run those checks. Additionally, config-time information does not show up in types, keeping the API simpler.

One key goal in using arrows to represent workflows is that they are amenable to *config-time* analysis. This is what we meant by *statically manifest* earlier: a statically manifest structure is suitable for analysis at config-time.

1.2 Contributions

- We leverage our new terminology of *config-time* and *process-time* phases for workflows to show how arrows, among other control structures, are a good fit for data science applications.
- A new approach to structuring composable effects with handlers, dubbed *Kernmantle*[3], is the main contribution of this work. This approach meets the design goals outlined above: it is a framework for statically analyzable composable tasks, executed via effect handlers (Section 4).
- We provide a number of practical examples of Kernmantle in action, including a lengthy, real-world case study. The development of Kernmantle was directly motivated by practical concerns, and we demonstrate that it solves practical problems. We hope that these examples will inspire other practitioners to use these techniques to build composable workflows (Section 5).

Though we focus here on using Kernmantle to build computations from arrows, Kernmantle does not rely on arrows being the underlying control structure, as we explore in Section 4.3.

Beyond these main contributions, our paper includes a comparison to related work (Section 6) and some thoughts on future directions (Section 7). All the work presented here is fully implemented. Our implementation is close to the excerpts we showed; we review the differences in the appendix.

2 Background

2.1 Arrows as tasks

Monads are popular models of effectful computations, but monadic computations are not amenable to config-time analysis of their structure. They simply prevent any distinction between config-time and process-time. Indeed, consider the type of the monadic bind:

$$(\gg) :: Monad\ m \Rightarrow m\ a \to (a \to m\ b) \to m\ b$$

The second parameter (the continuation) is a function whose argument is the result of running the first parameter—and we have to apply this function to learn anything about the final result. Accordingly, when effectful computations are structured as monads, the computational effects are intertwined with each other: it is impossible to extract one kind of effect without extracting all effects and the associated (pure) resulting value.

Hughes [2000] proposes instead to structure effectful computations around *arrows*, which, in our view, correspond neatly to the notion of a task in a data workflow; we use the terms *task* and *arrow* interchangeably.

In a nutshell, Hughes's idea is to eschew the Haskell function type and use a custom specialized task type $T\ a\ b$, representing a computation from an input type a to an output type b. We call such types of two arguments (be they tasks or not) *binary effects*, in contrast to effects encoded in monads or functors (of one type argument, the output only)

[3] https://github.com/tweag/kernmantle

which we call *unary effects*. Because the representation can be concrete (that is, built with routine algebraic datatypes), it is possible to extract config-time information even from the composition of two tasks ($f \circ g$), whereas this is impossible from ($g \gg\!= k$) or ordinary function composition.

Before we get to *Arrow*, we introduce its superclass, *Category*:

```
class Category t where
  id  :: t a a                  -- Identity
  (∘) :: t b c → t a b → t a c  -- Composition
```

We see that the (∘) operator we mentioned earlier is from *Category*. *Category* allows us to define a task that just returns its input (identity), and to sequence two tasks so that the output of the first becomes the input of the second (composition), resulting in a new task. Composition is therefore the most basic and omnipresent building block of our workflows. However, *Category* alone is not enough, as it does not satisfy our requirements for encoding tasks: we need extra operators to lift arbitrary calculations into tasks and pass previous results along the workflow. For that we use *Arrow* [4]:

```
class Category t ⇒ Arrow t where
  arr   :: (a → b) → t a b      -- Lift a pure function
  first :: t a b → t (a, c) (b, c)  -- Pass extra data along an
                                    -- Arrow for later use
```

This second property of passing data along a computation is often referred to as *strength*. We say that a category is *strong* if it supports it. Additionally, instances of *Category* and *Arrow* must also obey certain laws [Jacobs et al. 2009].

2.2 Haskell's arrow notation

We offer here a high-level intuition to GHC's arrow notation; Paterson [2001] has a more formal treatment. Readers familiar with this extension to Haskell can safely skip this subsection. Suppose we have three tasks *t1*, *t2*, and *t3*; all accept and produce *Int*s. Now, we want to write a task *t4*; *t4* should feed its input into *t1*, feed the constant 5 into *t2*, and then feed the sum of *t4*'s original input and the two intermediate results into *t3*, returning the output of *t3*. Using arrow notation, we can do this with the following code:

```
t4 = proc x → do y ← t1 ≺ x
                 z ← t2 ≺ 5
                 t3 ≺ x + y + z
```

The input to the new task t_4 is indicated with the **proc** keyword, which can be understood as a binding construct, much like a λ. Each command after the **do** (individual lines in arrow-notation are called *commands*) is read right-to-left: we state the input to a task, which is an ordinary expression; the task itself; and a pattern against which to match the answer—typically, just a variable to bind. If a task's result is to be ignored, the ← may be left out. The last line of a **proc** also omits the ←, because that task is the overall result of the **proc**.

Building up such a structure with the combinators in the *Arrow* class is a challenge. Here is what it would look like:

```
t4 = t3 ∘ arr (λ(z, (y, x)) → x + y + z) ∘ first t2
   ∘ arr (λyx → (5, yx)) ∘ first t1 ∘ arr (λx → (x, x))
```

The idea here is that tasks are chained together, using composition (∘). Using *first*, we can modify a task to work only on the first component of a pair, leaving the second component untouched. We thus must arrange for each task's argument to be the first component of an input pair, and to produce its result as the first component of the output pair. Accordingly, we must insert manipulators (embedded using *arr*) between the invocations of the tasks. Note that we start by embedding $\lambda x \to (x, x)$ because x is used as an input to the last task t_3.

The desugaring of arrow syntax handles all of the tupling, *arr* manipulators, and compositions seen in the explicit form above, letting us focus on the flow of data, not wrangling combinators.

Compare the arrow-notation form to its desugaring, above: in the desugaring, variables x, y, and z are brought into scope only in the manipulators embedded with *arr*—they are *not* in scope when naming a task. Similarly, in the arrow notation, variables bound by a **proc** or to the left of a ← are *not* in scope between the ← and the ≺. This is vital. It means that the choice of tasks to execute does *not* depend on the data operated on. It gives us the possibility of static, config-time analysis.

3 Basic tasks

Before we present Kernmantle, we show how a structure like *Arrow* can be used to model simple effects.

3.1 Pure Functions

Pure functions can be used as tasks[5]:

```
instance Category (→)
instance Arrow (→)
```

3.2 Kleisli

A monad is typically used to express the use of effects. It is thus convenient to embed a monadic computation into an arrow. This is the *Kleisli* construction:

```
newtype Kleisli m a b = Kleisli {runKleisli :: a → m b}
instance Monad m ⇒ Category (Kleisli m)
instance Monad m ⇒ Arrow (Kleisli m)
```

But wait: we have said that monads were trouble, because they cannot be analyzed at config-time. Accordingly,

[4] The presentation here is simplified from the definition in the *Control.Arrow* library module.

[5] We suppress less interesting parts of our implementation in this text. You can view the full definitions in our online implementation at https://github.com/tweag/kernmantle.

a good workflow design will minimize the use of *Kleisli*—essentially, each use of *Kleisli* creates a black box that cannot be inspected without data. However, given that the monad wrapped in the *Kleisli* arrow is exposed as a type parameter to *Kleisli*, that can be known. For example, it is easy to spot which tasks perform I/O, as they will have type *Kleisli IO*. It is, however, impossible to know precisely what I/O those tasks perform. The ability to lift any monad to an arrow has further practical importance: it guarantees that a framework based on arrows will always be able to reuse libraries using monads. Monad-based libraries are plentiful and we want to be able to leverage them.

3.3 Writer

The *WriterA* arrow allows an effect of writing information into a config-time channel—as long as that information does not depend on the input data. This could be useful for, e.g., *config-time* logging.

```
data WriterA w arr a b = WriterA w (arr a b)
```

This *WriterA* builds upon an underlying type *arr* (much like monad transformers do). If we specialize this underlying arrow to be a pure function, we see that *WriterA w* (→) *a b* is isomorphic to (*w*, *a* → *b*). That is, it is a function from input *a* to output *b*, along with something of type *w*—but that piece of data does *not* depend on the input *a*.

Consider, by contrast, the *Kleisli* arrow built around a regular *Writer* applicative functor, isomorphic to *a* → (*w*, *b*). Here, the written output of type *w can* depend on the input of type *a*. We thus prefer *WriterA* to enable config-time analysis.

As long as we can combine two values of type *w* and get a identity element for *w* (in other words, as long as *w* is a monoid), then *WriterA* is a arrow:

```
instance (Monoid w, Category t)
  ⇒ Category (WriterA w t)
instance (Monoid w, Arrow t)
  ⇒ Arrow (WriterA w t)
```

Given an existing arrow *t* :: *arr a b*, we can make it write at a side result of *x* :: *w* at config-time by writing *WriterA x t* :: *WriterA w arr a b*.

3.4 Cayley

It turns out that this *WriterA* arrow is just a special case of a more general type, the *Cayley* functor, after Rivas and Jaskelioff [2014]:

```
newtype Cayley f t a b = Cayley (f (t a b))
instance (Applicative f, Category t) ⇒
  Category (Cayley f t) where
    id                  = Cayley (pure id)
    Cayley p2 ∘ Cayley p1 = Cayley ((∘) ⟨$⟩ p2 ⟨*⟩ p1)
instance (Applicative f, Arrow t) ⇒
  Arrow (Cayley f t) where
    arr f               = Cayley (pure (arr f))
    first (Cayley p)    = Cayley (fmap first p)
```

Here, we are building on the notion of an applicative functor; we can think of these as monads, but where the bind operation *cannot* access the result of the previous computation. Interestingly, *Cayley* gives us exactly what we need to implement the *WriterA* arrow in terms of the *Writer* applicative functor—without losing the ability for config-time analysis. Here is the *Writer* applicative functor:

```
data Writer w a = Writer w a
```

Given that definition, we get the following isomorphisms:

```
Cayley (Writer w) arr a b ≈ Writer w (arr a b)
                          ≈ (w, arr a b)
```

This matches up exactly with our *WriterA w*, and so we choose to use *Cayley* (*Writer w*) instead. One could do the same for the *Reader* functor, which is just a pure function:

```
newtype Reader r a = Reader (r → a)
runReader :: Reader r a → r → a
runReader (Reader f) = f
```

Given that definition, we get the following isomorphisms:

```
Cayley (Reader r) (→) a b ≈ Reader r (a → b)
                          ≈ r → a → b
```

Given that the *Cayley* type can be used repeatedly to create several layers on top of an arrow, we are going to use a type operator as an alias to enhance readability:

```
type (↠) = Cayley
infixr 1 ↠
```

This allows us to write several layers of *Cayley* as follows:

```
Writer w ↠ Reader r ↠ Kleisli IO ≡
  Cayley (Writer w) (Cayley (Reader r) (Kleisli IO))
```

Even though we use an arrow-like symbol here, *Cayley* is definitely not function-like. For instance, layers cannot be reordered: (*Reader r* ↠ *Writer w* ↠ *eff*) *a b*, which expands to *r* → (*w*, *eff a b*), is not isomorphic to (*Writer w* ↠ *Reader r* ↠ *eff*) *a b*, which expands to(*w*, *r* → *eff a b*). The type *f* ↠ *g* should be read as "the binary effect *g* is conditioned by an effect in the functor *f*", meaning that an effect in *g* can be *built* or parameterized by an effect in *f*.

The key point of *Cayley* is its vast applicability: any process defined by an *Applicative* functor can be used to build an arrow. For example, in the Haskell ecosystem a growing number of parsing libraries (such as parsec, attoparsec, megaparsec, and optparse-applicative) provide an applicative functor interface. Using such an interface, an API can provide tasks of type *Parser* (*SomeArrow a b*) that reads some configuration from a file or command-line interface before building an actual computation. These tasks will still

be composable into a full workflow. Also, given that every *Monad* is also an applicative functor—because applicative functors do strictly less than monads—it means any monad library could be used to build such computations.

In sum, we can use a monad to build a task which performs computation at process-time (using *Kleisli*) or at config-time (using *Cayley*). Both options are available to the programmer and make sense in different contexts.

4 The Kernmantle framework

Having built up an intuition for tasks—and the fact that they can be composed and analyzed at config-time—we present our main contribution, the design of Kernmantle.

Essentially, Kernmantle is the composition of two well-studied functional programming techniques: control structures of two arguments like categories and arrows, and effect handlers. We have seen a thorough introduction to arrows, but let us now focus on effect handlers. The key idea behind an effect-handlers approach is that effectful computations are built up essentially as syntax trees. Then, a separate *interpretation function* traverses the syntax tree to perform the effects. Crucially, the syntax tree is just an ordinary data type, available for straightforward traversal. With that intuition—building up a syntax tree and then interpreting it to both analyze it and execute it—we dive into Kernmantle.

4.1 Overview

The key goal of Kernmantle is to allow programmers to chain together different tasks conveniently. These tasks might have different, even overlapping sets of effects, and we want to weave these effects together. Furthermore, once these tasks are combined, we need a way of both analyzing and executing the tasks. Critically, *analyzing* and *executing* are different: the analysis happens at *config-time*, looking for configuration mishaps or absent databases and data-processing happens at *process-time*. We thus need a way of *interpreting* a composed task, where the choice of interpretation can vary independently of the tasks being composed. We could choose, for instance, to have an interpreter for a production environment, with unrestricted I/O and available external databases; or, by contrast, an interpreter for a test environment, where results are pre-determined and read from a hard-coded hashmap. The tasks and the workflow itself would be fully shared between those two environments.

Before getting into the details, we first strive to give a high-level overview of the moving pieces. We will revisit all of these pieces in detail later, so some descriptions here will necessarily be a bit loose.

Task composition with (∘) combines two tasks of the same task type t. We thus need to identify the type t that all of our sub-tasks will be embedded into. This "master task" is a type we call *Rope*. Values of the *Rope* type are effectively a graph of atomic tasks, built up using some underlying control structure combinators (for instance, those of *Arrow*). A *Rope* is made out of two parts: a *core* and a *mantle*. The *core* exposes the underlying control structure. In the rest of our paper, the *core* will therefore most often have an *Arrow* instance. The *mantle* "surrounds" the *core* and is where the effects are tracked. Together, effect handlers must interpret all the effects in a mantle in terms of the effects allowed by the core. The handlers are interpretation functions, each of them converting a given effect to the *core* type, as a natural transformation. That is, they have the type $\forall\ x\ y.\ effect\ x\ y \to core\ x\ y$ and. This division between a *core* (underlying control structure) and a *mantle* (effects) is what gives Kernmantle and the main *Rope* type their name[6].

As we build up a *Rope*, the *core* often remains polymorphic: tasks will only require it to implement some class defining the underlying control structure they need. The *mantle* too will remain polymorphic: each task requires some effect handlers to be present in the mantle in order to be able to trigger these effects, but the tasks constrain neither the entirety of the handlers, nor their order within the mantle. It is only when *interpreting* our final workflow that we will instantiate both *core* and *mantle* to concrete types.

When interpreting, the core of the *Rope* will be set to a task type that is capable of modeling all of the effects used with the *Rope*. For example, if the *Rope* should be interpreted in an environment capable of I/O and which produces a side result of type *String*, then the core might be instantiated to something that contains a *Writer String* part and an *IO* part. The mantle is defined by a compile-time (or type-level) association list that maps effect names to discrete effect types. This association list allows each effect in the mantle to be referred to by name. Therefore, several instances of the same effect can coexist unambiguously. We can, for example, have multiple distinct effects writing to *String*s, because the effects can be distinguished by name. An effect paired with a name is called a *Strand*.

The effects in the mantle are *binary effects*, of kind $\star \to \star \to \star$. Despite the fact that a binary effect has the same kind as an arrow, it is a distinct concept. Binary effects do not (necessarily) have to be instances of the underlying control structure (such as *Arrow*). Here is a simple example:

```
data LogEff a b where     -- a logging effect
  LogEff   :: LogEff String ()
  LogFlush :: LogEff ()     ()
```

The *LogEff* constructor takes a *String* as input, and returns a trivial result. When this effect is used, the compiler checks that the input and output types match what is expected.

[6] In some outdoor sports like rock climbing or parachuting, a Kernmantle rope is a hybrid rope, composed of two parts made out of different fibers: the core which provides elasticity and resistance to the rope, around which is the mantle (or sheath) which protects the rope from abrasion and provides durability.

Along similar lines, *LogFlush* has neither input nor output. An interpreter for the *LogEff* effect must be able to handle both constructors.

To trigger an effect, we use the *strand* function, which takes the type-level name of a *Strand* and argument of type *effect a b* and returns a *Rope core mantle a b*, where *effect* is indeed an effect in the mantle. Accordingly, we can take any effect represented in the mantle, turn it into a *Rope*, and thus compose them—exactly what we want.

After the *Rope* is built, we must be able to interpret it. This is done by giving an interpretation function for each effect represented in the *Rope*'s mantle; these interpretation functions return an effectful computation, building off the core type of the *Rope*. In code, these interpretation functions are passed to the *handle* primitive, which takes a *Rope* with a non-empty mantle and returns a *Rope* with one fewer effect in its mantle. When all of the effects in the mantle are *handle*d away, we can execute the *Rope* in the effect type stored as the *core*.

4.2 The *Rope* type

The main Kernmantle datatype, *Rope*, is presented in Fig. 1. A *Rope* value is a closure storing the built-up composition of desired effects (such as logging). When given the appropriate *Handlers* (a heterogeneous list of effect handlers, whose types match up with the effect types in the *mantle*), the *Rope* can execute, returning an effect in the *core*.

Example workflow. An example of a *Rope* in action is in Fig. 2. This example is simple: it computes a result by chaining two sub-tasks, while logging its actions[7]. The logging effect is encoded with the *LogEff* type, and we provide a interpretation function *interpretLogEff* separately.

The workflow uses this effect under the name `"logger"` and uses the *strand* function (see Fig. 3) to embed the *LogEff* effect into the *Rope*. The `@"logger"` syntax is an example of a visible type application [Eisenberg et al. 2016]; it specifies the name of the desired strand. In order for the *strand* call to type check, we must know that `"logger"` :– *LogEff* is part of our mantle: the *InMantle* class (Fig. 3) asserts this requirement and provides a method *findHandler* that allows us to implement *strand*. Naming allows us to use different strands of effects with the same effect type. In this example, we could, for instance, have several logging channels, each one using the same *LogEff* type, but with different names. We also note that this workflow places a requirement on the *core*: that it should be an *Arrow*, so the whole *Rope* can be an *Arrow* too. Our use of arrow-notation desugars to use the combinators we defined as part of the *Arrow* class.

[7] The logging action here does depend on the data input into the task, and is not available for config-time analysis. *Some* effects will always depend on the data. We will see examples supporting config-time analysis shortly.

```
  -- The kind of our binary effects:
type BinEff = ★ → ★ → ★

  -- A Strand is a named binary effect; Symbol is
  -- a compile-time ("type-level") string:
data Strand = Symbol :– BinEff

  -- A Mantle is a list of named effects:
type Mantle = [Strand]

  -- Extract the effect type from a Strand:
type family StrandEff (t :: Strand) :: BinEff where
  StrandEff (_ :– eff) = eff

  -- A Handler is an interpreter for a named effect,
  -- interpreted in the core type:
type Handler (core :: BinEff) (strand :: Strand) =
  ∀ x y. StrandEff strand x y → core x y

  -- Handlers is a heterogeneous list of Handlers,
  -- including handlers for many Strands:
data Handlers (core :: BinEff) (mantle :: Mantle) where
  HNil  :: Handlers core '[]
  HCons :: Handler core strand
        → Handlers core mantle
        → Handlers core (strand : mantle)

  -- A Rope core mantle can be executed in a core type
  -- when given interpretations for the effects in mantle:
newtype Rope (core :: BinEff) (mantle :: Mantle) a b =
  Rope (Handlers core mantle → core a b)

  -- Rope c m is isomorphic to Cayley ((→) (Handlers c m)) c,
  -- so its instances will be the same as Cayley's
instance Category core ⇒ Category (Rope core mantle)
instance Arrow core ⇒ Arrow (Rope core mantle)
```

Figure 1. The *Rope* type, parameterized over a *core* and a *mantle*

Running the workflow. To run a *Rope* and execute its effects, we call the *handle* function in Fig. 4. The easiest way to understand *handle* is to look at its type. We see there that *handle* takes a way to interpret *eff* (the effect supported by the strand in question) in the *core* effect. In our running example, we interpret *LogEff* by executing it in a *core* of *Kleisli IO*. This interpretation function must support all possible input/output types from the effect, and hence *handle* has a higher-rank type [Peyton Jones et al. 2007]. Given the interpretation function, *handle* allows us to treat a *Rope* with the *eff* in its mantle as one without support for that effect. We can see how nested calls to *handle* can thus handle multiple effects. The core effect from the bare, mantle-less *Rope* can then be executed with a call to *strip*.

Here is how we actually execute the workflow:

```
  -- A binary effect type encoded as a GADT:
data LogEff a b where
  LogEff :: LogEff String ()
task1 :: Arrow t ⇒ t X Y
task2 :: Arrow t ⇒ t Y Z
  -- A workflow using this effect:
workflow :: (Arrow core, InMantle "logger" LogEff mantle)
         ⇒ Rope core mantle X Z
workflow = proc input → do
  a ← task1 ↢ input
  strand @"logger" LogEff ↢
    "Task1 completed, computed: " ++ show a
  b ← task2 ↢ a
  strand @"logger" LogEff ↢
    "Task2 completed, computed: " ++ show b
  id ↢ b
  -- We interpret our logging effect by printing
  -- each logged line to the console:
interpretLogEff :: LogEff a b → Kleisli IO a b
interpretLogEff LogEff = Kleisli putStrLn
```

Figure 2. A workflow with a logging effect

```
  -- InMantle name strandEff mantle says that effect eff
  -- is named name in the mantle:
class InMantle (name :: Symbol) (eff :: BinEff)
               (mantle :: Mantle) where
    -- Interpret a eff into core
  findHandler :: Handlers core mantle → eff x y → core x y
  -- InMantle instances exist that look up a handler in a mantle.
  -- Embed a named effect into a Rope
strand :: ∀ strandName strandEff mantle x y core.
          (InMantle strandName strandEff mantle)
       ⇒ strandEff x y → Rope core mantle x y
strand eff = Rope (λhandlers →
  findHandler @strandName @strandEff handlers eff)
```

Figure 3. The *strand* function which triggers effect execution, implemented thanks to the *InMantle* constraint

```
runWorkflow :: X → IO Z
runWorkflow x =
  let workflowWithoutEffs =
    handle @"logger" interpretLogEff workflow
  in case strip workflowWithoutEffs of Kleisli k → k x
```

This example can take advantage of Kernmantle's modularization: given that we offer an interpretation of effects independently of their definition, we can swap out one interpretation for another. Below, we collect the output using a *Kleisli* (*Writer* [*String*]):

```
  -- By supplying an effect handler, reduce the number
  -- of effects in a Rope's mantle:
handle :: ∀ name eff core mantle a b.
          (∀ x y. eff x y → core x y)
       → Rope core ((name :– eff) : mantle) a b
       → Rope core mantle a b
handle interp (Rope runner) = Rope (λhandlers →
  runner (HCons interp handlers))
  -- Take a Rope without a mantle and extract its core effect.
strip :: Rope core '[] a b → core a b
strip (Rope runRope) = runRope HNil
```

Figure 4. The *handle* function which is used to run an effect present in the mantle of a *Rope*

```
interpretLogEffWriter :: LogEff a b
                      → Kleisli (Writer [String]) a b
interpretLogEffWriter LogEff =
  Kleisli (λlogLine → Writer [logLine] ())
runWorkflowWithWriter :: X → ([String], Z)
runWorkflowWithWriter x =
  let workflowWithoutEffs =
    handle @"logger" interpretLogEffWriter workflow
  in case strip workflowWithoutEffs of
    Kleisli k → case k x of
      Writer logs result → (logs, result)
```

We choose to use *Kleisli* instead of *Cayley* precisely because we want the logged output to depend on the data. If we tried to write this last runner using *Cayley*, we would discover that we need to produce the logged output out of scope of the function argument that provides the data to log.

4.3 *Rope* need not be an *Arrow*

Having laid out *Rope*, we can now describe better how Kernmantle is independent of the choice of using *Arrow* to constrain the *core*. We need our underlying control structure to:

1. lift arbitrary calculations as tasks,
2. chain tasks together: express that a task should run before another,
3. pass the output of a previous task along the workflow so it can be reused later as the input of another task, and
4. allow any task to expose arbitrary config-time data, which can be analyzed before the process-time phase.

Points (1) and (2) are the bare minimum that a data workflow framework must provide. Point (3) is not always present: the popular Airflow framework does not allow directly passing data from one task to the next. Instead, it relies on serialization to disk to communicate between tasks. Point (4) is to our knowledge the main novelty brought by our approach to the domain of data workflows. Note that while these first three requirements are widespread and match most data applications, some applications could have fewer requirements and some could have more. Arrows address all these points and benefit from syntactic support in GHC via the Arrow notation [Paterson 2001]; this is why we consider them as the default control structure in which to present our proposal. Our proposal is orthogonal to this choice, as Kernmantle is parametric in a typeclass *T* such that:

- *T* has a single type parameter of kind $\star \to \star \to \star$,
- the *Cayley* functor satisfies *T*, recalling that *Rope* is isomorphic to *Cayley* ((→) *Handlers core mantle*) *core*. In other words, we must be able to declare an **instance** (*Applicative f*, *T x*) ⇒ *T* (*Cayley f x*) for all *f* and *x*.

The choice of the underlying control structure determines the shape of the graph of tasks that can be built with a workflow. For example, we could use *ArrowChoice* instead of *Arrow* as the underlying structure of *Rope* to allow conditional branching. We will discuss this in Section 7.2.

5 Practical examples with Kernmantle

Having seen a toy example of how algebraic effects with arrows works, we now explore several practical examples, showing how this combination of foundational ideas enables compositional tasks amenable to config-time analysis.

5.1 A workflow with abstract data sources

The first effect, *ReadResource*, corresponds to reading an external resource, inspectable at config-time:

```
type ResourceId = String   -- A name to identify a resource
data DataSet where ...
data ReadResource a b where
  ReadDataSet :: ResourceId → ReadResource () DataSet
  ReadRawFile :: ResourceId → ReadResource () ByteString
  ...
```

This type allows us to read all sorts of resources, either as structured data—whose type is known in advance—or as raw bytestrings.

Setting up our effect this way—where the *ResourceId* does not appear in the first parameter of the *ReadResource* constructors—imposes a constraint: the resource identifier must be knowable *at config-time*. Because the choice of *ResourceId* is required in order to construct the binary effect, it will not have access to process-time data; recall that process-time data, as the input to a task, is not in scope when building the task itself. Here is how it would work in practice, to define the following workflow that accesses *ResourceId*s `"dataset1"` and `"dataset2"`:

```
joinDataSetsOnColumn ::
  Arrow t ⇒ t (DataSet, DataSet, String) DataSet

workflow :: (Arrow core
            , InMantle "resources" ReadResource mantle)
         ⇒ Rope core mantle () DataSet
workflow = proc () → do
  dataSet1 ← readDataSet "dataset1" ⤙ ()
  dataSet2 ← readDataSet "dataset2" ⤙ ()
  joinDataSetsOnColumn ⤙ (dataSet1, dataSet2, "Date")
  where
    readDataSet identifier =
      strand @"resources" (ReadDataSet identifier)
```

This workflow loads two data sets and joins them according to their "Date" column. As before, we now need a way to interpret this workflow. We must be able

- to find—at config-time—all the identifiers of the resources it requires, and
- to feed it—at process-time—the contents of the resources it requires.

Thus, when we interpret this workflow, we need to instantiate *core* with a type that can be split into two parts: one relevant at config-time and one relevant at process-time. Here is the desired type of the interpreted workflow:

```
interpretedWorkflow ::
  (   -- Requirements collected at config-time:
   Set ResourceId
     -- Process-time function taking the final mappings:
  , Map ResourceId FilePath → IO DataSet)
```

In this concrete setting, we are expecting that resources correspond to *FilePath*s, but this is not necessary—resources can be gathered from a variety of sources.

The key detail here is that the interpreted workflow has two separate components. If we wish to do only a config-time analysis, not to run the workflow, we can just project out the first component of the tuple. If we want to run the workflow, however, we use the first config-time component to set up the argument to the process-time function. This set-up might involve looking in a configuration file, or parsing command-line arguments, for example.

Interpretation always produces a result in the *core* of a *Rope*—which in our case is an arrow—and the type above does not appear to be an arrow. Yet *interpretedWorkflow* is isomorphic to the following type:

```
(  Writer (Set ResourceId)
↠ Reader (Map ResourceId FilePath)
↠ Kleisli IO) () DataSet
```

Here, we are using the suggestive ↠ notation instead of *Cayley*. The implementation of this interpretation function follows:

```
interpretedWorkflow =
  case strip (handle @"resources"
    interpretReadResource workflow2) of
  Cayley (Writer resources (Cayley (Reader run))) →
    (resources
    , λmapping → case run mapping of
        Kleisli action → action ())

interpretReadResource
    :: ReadResource a b
  → (   Writer (Set ResourceId)
      ↠ Reader (Map ResourceId FilePath)
      ↠ Kleisli IO) a b
interpretReadResource (ReadDataSet ident)
    = Cayley $ Writer (singleton ident) $
      Cayley $ Reader $ λmapping → Kleisli $
        λ() → loadDataSet (mapping ! ident)
interpretReadResource (ReadRawFile ident) = ...
```

We can yet make this approach even more expressive. We want to add the possibility of including help text that describes the resource and an optional default *FilePath* of where to find the resource. To do so, we replace the collected *Set ResourceId* with a *Map ResourceId* (*String*, *Maybe FilePath*). Given that this pattern arises whenever we do config-time analysis of resources[8], we introduce a *Config* newtype that encapsulates it:

```
newtype Config k v a =
  Config (Writer (Map k (String, Maybe v))
                 (Reader (Map k v) a))
```

This *Config* type exactly matches up with our desired interpretation of *ReadResource* above. It writes out a mapping from its key type *k* to pairs containing help text and an optional default value, and its second component operates in an environment containing a mapping from keys to values. With *Config*, we can re-express our *core* type as

```
type CoreEff = Config ResourceId FilePath ↠ Kleisli IO
```

and *interpretedWorkflow* gets the following type:

```
interpretedWorkflow ::
  (Config ResourceId FilePath ↠ Kleisli IO) () DataSet
```

To summarize, this *CoreEff* type supports usage scenarios focusing either on config-time or process-time. At config-time, we can

1. accumulate all the resources required by a workflow, along with optional default paths and help strings to describe them;
2. generate default configuration files, a command-line help page, etc.;
3. stop there, ignoring the process-time component.

If we wish to actually run the workflow, we can

1. accumulate all the resources as above to retrieve default paths;
2. load configuration files, parse command-line etc. to override the default paths;
3. check if the final configuration is complete and valid;
4. run the workflow by feeding it the validated configuration.

The ability to extract the *ResourceId*s in advance of running the workflow is crucial: it allows our program to examine its environment at config-time to determine whether some error may happen at process-time. If any required resource is unavailable, then the application will be stopped. Perhaps even the data in these data sets can be pre-fetched, witnessing the possibility of using the Kernmantle framework to enable more eager validation or optimization.

5.2 A workflow of cacheable tasks

If a task is run twice with the same input, we want to be able to reuse cached results[9]. Indeed, the validity of caching is one of the main advantages of working in a pure language, where we can identify code that is free of side effects. Assume an input type *A* that can be serialized and compared for equality (we say *A* can be *hashed*) and an output type *B* that can be serialized and deserialized. Given a task of type *Rope core mantle A B*, perhaps part of a bigger workflow, this task should be able to store its results, index them with values of type *A* and return the cached values when appropriate. We can design a way to add caching to any task fulfilling these requirements:

```
caching :: ... ⇒ Maybe String
        → Rope core mantle a b → Rope core mantle a b
```

The *caching* method wraps a task and optionally gives it a name. The name can disambiguate tasks that have the same input and output types, but perform different actions.

We leverage the fact that a task, besides requiring some effects to be present in the mantle, can also place requirements on the core. We have the following class that can constrain our *core*:

```
class ProvidesCaching core where
  withStore :: (Hashable a, Serializable b)
            ⇒ Maybe String → core a b → core a b
```

Given a *Store* type acting as a *Map InputHash ByteString* (where the values are the serialized results of tasks) persisted on disk, we can implement this for *Reader Store* ↠ *Kleisli IO*:

[8] In fact, it is common enough to be the foundation of the porcupine library.

[9] This example of usage of the Kernmantle architecture is embodied in the funflow library.

```
instance ProvidesCaching (Reader Store ↠ Kleisli IO)
```

Our wanted *caching* function can now be implemented for any *Rope*, *regardless* of the effects in its mantle, by just placing a *ProvidesCaching* constraint on its *core*:

```
caching :: (ProvidesCaching core, Hashable a, Serializable b)
        ⇒ Maybe String
        → Rope core mantle a b → Rope core mantle a b
caching mbName (Rope f) = Rope $ λhandlers →
  withStore mbName (f handlers)
```

We can easily propagate *ProvidesCaching* through *Cayley*:

```
instance {−# overlapping #−}
  (Functor f, ProvidesCaching eff) ⇒
  ProvidesCaching (f ↠ eff) where ...
```

Having this instance can be useful in applications where the *core* stacks additional *Cayley* wrappers around a *Kleisli* arrow. We have shown a use for such a *core* in section 5.1.

Naming tasks. The choice to optionally name tasks in *caching* arises from an unusual interplay between the nature of workflows and how they are used. The need to name the tasks comes from the desire not only to cache results within a single run of the workflow, but also across different runs. If we have two distinct tasks t_1 and t_2, both consuming and producing *Double*s, we want to make sure that the cache for t_1 is not used when processing t_2. If we wanted to support only caching within one run of the workflow, we could accomplish this by, say, making a temporary file for each of t_1 and t_2 and storing their caches there. We could even store the cache purely in memory, by creating a new *Store* for every call to *caching*.

Even if we wanted to support inter-process caching, we could avoid names by recording an invented name in a configuration file. However, recall that workflows tend to be written and then operated by domain experts—there is no divide between developer and user, here. This means that the expert may run a workflow, decide on a different analysis, re-compose the workflow (but reusing t_1 and t_2 in the new version), and then re-run the workflow. Indeed, such patterns are common in practice. By allowing for named caches, even when the executable binary changes, the cache can be reused between runs.

Why, then, make the naming optional? Because a name can be deterministically derived from the DAG comprising the workflow. This DAG can be observed at config-time, and thus names can be derived for each task from its location in the DAG. Using this automatic naming means that caches would be invalidated whenever the binary changes, but it avoids the needs for programmer-supplied names or configuration files. The design above, using *Maybe String* for the name, supports a hybrid model, where some caches are persistent across binary changes, and others are not.

5.3 Case study: a workflow for computational biology

After seeing several effects used separately, let us use them in a single workflow that needs all of them to implement an actual use case that arose in practice. In this section, we first review the requirements that our workflow solution must satisfy, then we will see how the Kernmantle framework can meet these needs, using the effects we have already seen.

A basic workflow in computational biology must meet these *functional* requirements:

1. Use an existing biochemical model, which refers to some default parameters and initial conditions
2. Allow for specification/override of parameters and initial conditions
3. Run the simulation, perhaps via an external tool, gathering results
4. Write the results in a standard serialization format

On top of these functional requirements we have the following extra *non-functional* requirements. They do not affect the computed results, but they enhance the workflow of the biomodelers:

1. The workflow should be able to cache intermediate results, in order to improve performance.
2. Parameters and initial conditions should be retrievable from configuration files, or the command-line, or other sources (spreadsheets, network, etc.). This allows biomodelers to easily explore the parameter space to find the most realistic results.
3. Biomodelers should be able to embed arbitrary functions as tasks, instead of using a pre-defined set of available tasks. These custom tasks should also be cacheable.
4. Tasks should be self-contained, reusable, and shared between several modeling projects workflows.
5. A workflow of tasks should be platform-independent.
6. A workflow should be able to reuse the same task in multiple places, without conflict; each occurrence of the same task should have access to its own parameters. This requires a *namespacing* facility.

We showed earlier in the paper how we can deal with the non-functional requirements from 1 up to 5:

1. Caching is covered in Section 5.2.
2. Reading configuration files is covered in Section 5.1.
3. Arbitrary functions can be encoded via *arr*::*Arrow* $t \Rightarrow (a \to b) \to t\ a\ b$, and such a task would be compatible with the caching mechanism we have explored.
4. Reuse is straightforward, as tasks are compositional. Caches are retained between reuses via cache names.
5. Platform-independence is achieved through the lookup of, e.g., *ResourceId*s to get *FilePath*s; we do not hard-code file paths into our workflows.

We are left to explore non-functional requirement 6, as well as the functional requirements.

We first generalize our *ReadResource* effect to an *AccessResource* effect, which also permits *writing* resources. Additionally, we need two new effects. The first new effect allows to trigger the simulation of our biochemical model:

```
data Biomodel params result where ...
```

Why do we want to make *Biomodel* a full-fledged effect and not just provide some task—implemented with *AccessResource*—that would solve a *Biomodel*? For flexibility. We can conceive of several different ways to interpret the simulation of a model. For example, we might have a straightforward pure function that computes results from parameters, or we could have one that produces information for tracing and debugging purposes, or we could have one that writes out visualizations of its intermediate states, or even invoke an external tool to do the core work.

The second effect we need is commanded by non-functional requirement 6. We need a way to *namespace* the tasks in our workflow so that the *AccessResource* effects can use that namespace to generate qualified default paths. Just like for caching in section 5.2, we are going to use a class providing a method to wrap a *Rope*:

```
  -- components of a name, like
  -- "Model1.Proteins.ForwardSim":
type Namespace = [String]
class Namespaced eff where
  addToNamespace :: String → eff a b → eff a b
instance Namespaced core
  ⇒ Namespaced (Rope core mantle) where ...
```

Here is an example of such a workflow, running two simple cacheable models:

```
vanderpol :: Biomodel Double ()
chemical  :: Biomodel Double ()

  -- access a resource
access :: ResourceId → String → Maybe a
       → AccessResource () a

workflow
  :: (Arrow core, ProvidesCaching core, Namespaced core
    , InMantle "logger" LogEff mantle
    , InMantle "bio" Biomodel mantle
    , InMantle "resources" AccessResource mantle)
  ⇒ Rope core mantle () ()
workflow = proc () → do
  strand @"logger" LogEff ≺ "Beginning workflow"
  addToNamespace "vanderpol" (proc () → do
    mu ← acc "mu" "mu parameter" (Just 2) ≺ ()
    caching Nothing (strand @"bio" vanderpol) ≺ mu
    ) ≺ ()
```

```
type CoreEff =
      -- Get the namespace we are in. As it is first, it can
      -- parameterize all the subsequent layers:
    Reader Namespace
      -- Accumulate all the wanted options. The final option
      -- names are determined by the current namespace,
      -- as well as the default file paths (which are exposed
      -- as options, too):
  ↠ Config String Dynamic
      -- Accumulate the context needed to know what
      -- to take into account to perform caching:
  ↠ Writer CachingContext
      -- Get a handler to the content store, to cache
      -- computations:
  ↠ Reader Store
  ↠ Kleisli IO   -- Perform I/O
```

Figure 5. The core effect for the biomodeling case study

```
  addToNamespace "chemical" (proc () → do
    k ← acc "k" "Reaction rate" (Just 2) ≺ ()
    caching Nothing (strand @"bio" chemical) ≺ k
    ) ≺ ()
strand @"logger" LogEff ≺ "Workflow finished"
where
  acc n d v = strand @"resources" (access n d v)
```

Our two models here do not produce any result to feed into later tasks. Instead, our interpretation function for *Biomodel* effects will always write the raw result to disk, along with a visualization of the running biochemical simulation; it is elided here for space.

We have one question left: what *core* should we use to interpret this *workflow*? The answer lies in our list of non-functional requirements. Requirements 1 and 3 tell us that each task may need to access a store to put cached results in. Besides, 3 tells us that any custom task should be cacheable. However, given that these tasks can internally require some options, if an option changes we should invalidate the cache. Consequently we need some way for a task to expose to a possible encompassing *caching* call to all the options that may affect the hash function used, via a *CachingContext*. Requirement 2 tells us that we need to be able to perform *IO*. Finally, 6 tells us that each task must be made aware of the namespace it is in. It also tells us that the interpretation of both file accesses and option effects should be conditioned on the namespace. This gives us the core effect in Fig. 5.

A *main* action including the calls to *handle* to build such an effect is in our online implementation[10].

[10] https://github.com/tweag/kernmantle/blob/master/odes/examples/BiomodelUseCase.hs#L478

```
type Rope' effects = FreeA (Sums effects)
newtype FreeA t a b = FreeA
  (∀ arr. (Arrow arr) ⇒ (∀ x y. t x y → arr x y)
                      → arr a b)
data (:+:) t1 t2 x y = INL (t1 x y) | INR (t2 x y)
data VoidA x y   -- no constructors
type family Sums (ts :: [Type → Type → Type]) where
  Sums '[]     = VoidA
  Sums (t : ts) = t :+: Sums ts
```

Figure 6. Free arrow of sum of binary effects

If biomodelers then want to fully run the pipeline, config-time will happen first: they will see on the terminal a list of potential errors and warnings (if a configuration file is incorrect, if a resource to read is mapped to a non-existent file, etc.). Then, if there were no errors, a summary of the final validated configuration is printed for debugging purposes and the execution switches to process-time, at which point every biomodel simulation and every file access will trigger a log line so they can follow what happens.

6 Related work

6.1 Roots of *Rope*

Our inspiration for the *Rope* type comes from the effect handlers literature [Kiselyov and Ishii 2015; Plotkin and Pretnar 2013]. The *Rope* type is roughly a final encoding of a free arrow whose generator is a sum of several effects; see Fig. 6.

The idea of structuring computational effects around monads is due to Moggi [1991]. This idea then spread among programming languages. Unfortunately, monads do not compose well, and at worst one monad needs to be constructed for each possible combination of effects.

An elegant way to construct such combinations is based on *algebraic effects* [Plotkin and Pretnar 2013]. The idea is to interpret every computational effect by a handler, generalising the notion of an exception handler. A particularly fruitful realization of effects and handlers is the *Free* monad [Kiselyov and Ishii 2015]. The idea of a free construction is to add the operations of a given algebraic structure (in this case, a monad) around a set of generators (which need not exhibit the structure). The free monad can be defined this way:

```
FreeM f a = ∀ m. Monad m ⇒ (∀ x. f x → m x) → m a
```

The f type parameter can be any functor. Because the sum of two functors is also a functor, we recover compositionality: a neat generalization of both *FreeM f* and *FreeM g* is *FreeM* $(f :+: g)$[11].

For reasons explained in our introduction, we choose instead to structure computation around arrows, and thus use an *Arrow* constraint instead of *Monad*. m is just renamed to t and changed to take two parameters, obtaining the free arrow (*FreeA* in Fig. 6). Accordingly, the free arrow of a sum of effects (*Rope'*) is in fact very close to Kernmantle's *Rope*. The differences are as follows:

Names We give names to the different cases of the sum, thanks to compile-time symbols. This means that one can use the same effect several times, without ambiguity. Each operand of the sum is a *Strand*, a named effect.

Using a record We distribute the sum over the arrow, using the isomorphism $(a :+: b) \to c \approx (a \to c, b \to c)$. This means that, instead of a named sum, we have a named product, also called a *record*. Such records benefit from more language and/or library support than named sums [Kiselyov et al. 2004], which we can reuse. Each item in the record is a *Handler*, and the record itself is our *Handlers*.

Containing a core We do not universally quantify over the target arrow but rather make it a parameter of *Rope* (where it is called *core*) for the reasons exposed in 4.3.

6.2 Algebraic effects for non-monadic structures

Lindley [2014] and Pieters et al. [2017] also study the integration of algebraic effects with arrows and other non-monadic structures. Their approach, however, is dual to ours: they use a language that has first-class support for algebraic effect handlers (e.g. via delimited control operators), and they show how to build arrows or other structures from these effects.

In our proposal, we build effects from pre-existing binary structures (categories, profunctors, strong profunctors, profunctors with choice, etc). This design emphasizes reusability of pre-existing control structures, depending on the needs of the application. Tasks are able to place a minimal set of requirements on their core, expanding these requirements if needed. Thus, a full workflow of tasks could be built only on top of *Category* if none of its subtasks required more structure. Crucially, the whole API remains unchanged, as it is agnostic of the chosen control structure: as long as *Cayley* can propagate an instance of that control structure, *Rope* simply propagates the structure provided by the core. One could successfully argue that this design makes our work a bit more Haskell-centric, but the same could be said of the body of work regarding monadic algebraic effects, like monadic effects. We believe the applicability of our work goes farther than Haskell: few languages provide first-class effect handlers, while control structures as monads, categories, and arrows have become more widespread.

6.3 Comparison with existing workflow tools

There exist broadly two types of workflow tools: those expressing eagerly evaluated tasks that must terminate before the next one goes on (Luigi, Airflow) and those expressing ongoing data transformations that run forever in parallel to process incoming batches or streams of data (Storm, Beam). Kernmantle could be used as a basis for both types. Indeed,

[11] The popular polysemy library exhibits this structure.

even if we only show uses of eagerly evaluated tasks in our examples for the sake of simplicity, *Rope* does not enforce any execution order between the tasks. When we have a composition $b \circ a$ of two tasks, it just links together the output of a with the input of b. Whether effects in a are fully executed before effects of b start, or whether a and b's effects could run in parallel is totally up to the *core* that has been selected. If this core is *Kleisli IO*, then we inherit the strict nature of *IO*, *unless* our interpretation functions use *unsafeInterleaveIO* or explicitly fork some threads. Or we use can types from the pipes/conduit/streaming ecosystems in Haskell as our *core*. Kernmantle is a *bare* workflow framework that provides structure to applications or to libraries thanks to concepts that are rooted in formal computer science (as arrows are) that can be used to implement the features of existing tools, and provide config-time facilities that go beyond their capabilities.

6.4 Build systems

There is a clear correspondence between workflows, as we describe here, and a certain class of build systems [Mokhov et al. 2018]. In many build systems, a dependency graph is first constructed, and then tools are run to create build targets. The dependency graph phase corresponds to our config-time phase. Running tools corresponds to running our tasks in the process-time phase. Additionally, a crucial feature of build systems is caching, which can be supported as in Section 5.2. In fact, as far as we can see, our workflows with caching can be used as build systems out-of-the-box, provided that running tools can be expressed as tasks (i.e., their effects are captured by the input and output parameter of the task type).

6.5 Comparison with applicative functors

As we mentioned, applicative functors also provide a separation between configuration and processing. For instance, optparse-applicative uses prior knowledge of what it will parse to generate help pages. If we were to express Kernmantle over applicative functors, we must notice that tasks of type $input \longrightarrow F\ output$, where F is an applicative functor, lose all capacity to inspect F at config-time. Consider then task types of the form $F\ (i \longrightarrow M\ o)$, where M is a monad—most often containing *IO* so actual work can be performed at process-time. This approach means that part of the abstraction leaks: the M monad is exposed directly to the user, and therefore they could directly perform side-effects by targeting M instead of the effects of the mantle. Binary effects encapsulate this monad nicely: users are presented only with some type $T\ i\ o$; they are insulated from the innards of T.

The second consideration is syntactic. In our experience, arrow notation is well understood by data scientists: it captures the familiar notion of workflow, without referring to underlying abstractions. Existing syntactic sugar for applicatives does not capture as well the intuition of a workflow because it really expresses parallel composition rather than sequential composition.

Finally, arrows offer a finer-grained control over the available control operators, given the wealth of existing classes to work with them.

7 Discussion and future work

7.1 Current limitations of Kernmantle

The utility of the framework hinges on finding the right *core* type for a use case. We showed in Section 5.3 that this core type can (and should) be derived directly from the requirements of our application, composing it out of *Kleisli* and *Cayley* layers. This step requires some care, but we can mitigate this by remembering that, in a real-world application of Kernmantle, the person in charge of determining the effects/implementing them (a developer) and the person using these effects in a workflow (a data scientist) do not need to be the same person. They thus do not require the same set of skills, as Kernmantle supports the separation of concerns.

Contrary to some existing workflow tools (like Luigi, Spark or Beam), Kernmantle does not allow subtasks of a workflow to run over several workers. Haskell binary code is indeed much less portable than Python or Java bytecode. Previous work allows us to use Haskell in a distributed environment [Epstein et al. 2011], not by shipping code to be executed by workers but by shipping closures containing a reference (assuming all workers are running the same executable) to the function to execute as well as its serialized inputs. The GHC *StaticPointers* extension, and distributed-closure and sparkle packages have been developed to address that need. Our appendix contains a plan to support distributed execution without the need for distributed closures, by directly relying on the static nature of the workflow itself.

7.2 The overall graph structure of a workflow

The bulk of this paper has been mostly focusing on a very static DAG structure, as it has been sufficient for us to implement our examples and use case. However, Kernmantle is not restricted to this. In the end with the right underlying control structure it is possible to get the following:

Branches. Selecting which task to perform next based on some input data can be provided with *ArrowChoice* (or *Choice* from profunctors). Note that this approach does not necessarily make the graph dynamic: in most cases all the possible branches can still be known at config-time, but they will not all be taken at process-time.

Cycles. There are two ways we can think of cycles in a workflow: either (A) tasks that recursively call themselves until some condition is reached, or (B) continuously running tasks which sometimes feed their output back into their inputs. We did not make use of continuously running tasks

in this paper (which could be implemented by selecting a specific *core*, see Section 6.3). Instead, all the tasks in our examples had their effects eagerly evaluated before the next task proceeds—which necessitates that all of these tasks terminate.

In the case of (A), any task can recursively call itself without needing anything more than *Category*, but any practical use case is likely to require *ArrowChoice* in order to detect a terminal case. Case (B) is supported by *ArrowLoop*, but is likely to be useful only with *core*s that implement continuous, parallel execution of tasks, since any task returned by *loop* will never halt. This type of construction is heavily used in signal processing—for instance, to add reverberation to a signal. In a data science context, this could be used to implement a task which is fed a dataset in a streamed fashion and outputs a new dataset, fed back and merged with its input.

Data structure traversal. Functions *mapM* and *traverse* apply actions with side effects to each element of a collection. The profunctors package provides a generalization of these primitives: the *Traversing* control structure. This allows us to traverse a *Traversable* with a task, and even to traverse arbitrary data structures with *wander* if we have optics (lenses or traversals) for them. We envision this to be very useful in data science applications.

8 Conclusion

We have explored the Kernmantle architecture, where workflows comprise composable tasks. Each task can refer to arbitrary effects, which can be interpreted in a modular manner. This design enables *config-time analysis*, important when a workflow is being composed and run by domain experts. This underlying architecture is extensible, and already works on non-trivial examples.

Acknowledgments

We thank our colleagues at NovaDiscovery, Tweag, and the anonymous reviewers, all of whom helped develop these ideas and improved this text.

Bernardy is supported by grant 2014-39 from the Swedish Research Council, which funds the Centre for Linguistic Theory and Studies in Probability (CLASP) in the Department of Philosophy, Linguistics, and Theory of Science at the University of Gothenburg.

This material is based upon work supported by the National Science Foundation under Grant No. 1704041. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation.

References

Martin Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A System for Large-Scale Machine Learning.. In *OSDI*, Vol. 16. 265–283.

Baldur Blöndal, Andres Löh, and Ryan Scott. 2018. Deriving via: Or, How to Turn Hand-Written Instances into an Anti-Pattern. In *Proceedings of the 11th ACM SIGPLAN International Symposium on Haskell* (St. Louis, MO, USA) *(Haskell 2018)*. Association for Computing Machinery, New York, NY, USA, 55–67. https://doi.org/10.1145/3242744.3242746

Ronan Collobert, Samy Bengio, and Johnny Mariéthoz. 2002. *Torch: a modular machine learning software library.* Technical Report. Idiap.

Richard A. Eisenberg, Stephanie Weirich, and Hamidhasan Ahmed. 2016. Visible Type Application. In *European Symposium on Programming (ESOP) (LNCS)*. Springer-Verlag.

Jeff Epstein, Andrew P. Black, and Simon Peyton-Jones. 2011. Towards Haskell in the Cloud. In *Proceedings of the 4th ACM Symposium on Haskell* (Tokyo, Japan) *(Haskell '11)*. Association for Computing Machinery, New York, NY, USA, 118–129. https://doi.org/10.1145/2034675.2034690

M Erdmann, M Rieger, B Fischer, and R Fischer. 2017. Design and Execution of make-like, distributed Analyses based on Spotify's Pipelining Package Luigi. In *J. Phys. Conf. Ser.*, Vol. 898. 072047.

Chris Heunen and Bart Jacobs. 2006. Arrows, like monads, are monoids. *Electronic Notes in Theoretical Computer Science* 158 (2006), 219–236.

Paul Hudak, Antony Courtney, Henrik Nilsson, and John Peterson. 2003. *Arrows, Robots, and Functional Reactive Programming.* Springer Berlin Heidelberg, Berlin, Heidelberg, 159–187. https://doi.org/10.1007/978-3-540-44833-4_6

John Hughes. 2000. Generalising monads to arrows. *Sci. Comput. Program.* 37, 1-3 (2000), 67–111. https://doi.org/10.1016/S0167-6423(99)00023-4

Bart Jacobs, Chris Heunen, and Ichiro Hasuo. 2009. Categorical semantics for arrows. *Journal of functional programming* 19, 3-4 (2009), 403–438.

Oleg Kiselyov and Hiromi Ishii. 2015. Freer monads, more extensible effects. In *Proceedings of the 8th ACM SIGPLAN Symposium on Haskell, Haskell 2015, Vancouver, BC, Canada, September 3-4, 2015*. 94–105. https://doi.org/10.1145/2804302.2804319

Oleg Kiselyov, Ralf Lammel, and Keean Schupke. 2004. Strongly typed heterogeneous collections. In *Haskell '04: Proceedings of the ACM SIGPLAN workshop on Haskell*. ACM Press, 96–107. http://dx.doi.org/http://doi.acm.org/10.1145/1017472.1017488

Michael Kotliar, Andrey V. Kartashov, and Artem Barski. 2019. CWL-Airflow: a lightweight pipeline manager supporting Common Workflow Language. *GigaScience* 8, 7 (07 2019). https://doi.org/10.1093/gigascience/giz084 arXiv:https://academic.oup.com/gigascience/article-pdf/8/7/giz084/28954484/giz084.pdf giz084.

Sheng Liang, Paul Hudak, and Mark Jones. 1995. Monad Transformers and Modular Interpreters. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Francisco, California, USA) *(POPL 1995)*. Association for Computing Machinery, New York, NY, USA, 333–343. https://doi.org/10.1145/199448.199528

Sam Lindley. 2014. Algebraic effects and effect handlers for idioms and arrows. In *Proceedings of the 10th ACM SIGPLAN workshop on Generic programming*. 47–58.

Conor McBride and Ross Paterson. 2007. Applicative programming with effects. *Journal of Functional Programming* 18, 01 (2007), 1–13. https://doi.org/10.1017/S0956796807006326

Eugenio Moggi. 1991. Notions of computation and monads. *Information and computation* 93, 1 (1991), 55–92.

Andrey Mokhov, Georgy Lukyanov, Simon Marlow, and Jeremie Dimino. 2019. Selective applicative functors. *Proceedings of the ACM on Programming Languages* 3, ICFP (2019), 1–29.

Andrey Mokhov, Neil Mitchell, and Simon Peyton Jones. 2018. Build systems à la carte. *Proceedings of the ACM on Programming Languages* 2, ICFP (2018), 1–29.

Ross Paterson. 2001. A new notation for arrows. *ACM SIGPLAN Notices* 36, 10 (2001), 229–240.

Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Mark Shields. 2007. Practical type inference for arbitrary-rank types. *Journal of Functional Programming* 17, 1 (Jan. 2007).

Simon Peyton Jones, Stephanie Weirich, Richard A. Eisenberg, and Dimitrios Vytiniotis. 2016. A reflection on types. In *A list of successes that can change the world*. Springer. A festschrift in honor of Phil Wadler.

Ruben P Pieters, Tom Schrijvers, and Exequiel Rivas. 2017. Handlers for non-monadic computations. In *Proceedings of the 29th Symposium on the Implementation and Application of Functional Programming Languages*. 1–11.

Gordon D. Plotkin and Matija Pretnar. 2013. Handling Algebraic Effects. *Logical Methods in Computer Science* 9, 4 (2013). https://doi.org/10.2168/LMCS-9(4:23)2013

Exequiel Rivas and Mauro Jaskelioff. 2014. Notions of Computation as Monoids. *CoRR* abs/1406.4823 (2014). arXiv:1406.4823 http://arxiv.org/abs/1406.4823

Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, Ion Stoica, et al. 2010. Spark: Cluster computing with working sets. *HotCloud* 10, 10-10 (2010), 95.

A Further examples

These next two examples show a slight generalization of *handle* that allows one effect to be interpreted in terms of others.

A.1 Loading arbitrary configuration via dynamic typing

We can go further, allowing the user to request all necessary configuration—not just resource identifiers mapped to *FilePath*s—by introducing a new effect:

```
data GetOpt a b where
  GetOpt :: (Show v          -- Allows to print the default value
                             -- or write to configuration files
           , Typeable v)     -- Allow us to use dynamic typing
         ⇒ String            -- A name for the option
         → String            -- A help string
         → Maybe v           -- A possible default value
         → GetOpt () v
```

This effect can be interpreted in the same way as *ReadResource*, using the same logic as *interpretReadResource*, by generalizing our *core* type a little bit:

```
type CoreEff = Config String Dynamic ↠ Kleisli IO
```

Because our configuration values can be of any type, we need some dynamic typing here. For example, the `"num-iterations"` option will be an *Int*, but the `"graph-title"` will be a *String*. Haskell's *Dynamic* type works with the *Typeable* constraint in *GetOpt* and supports run-time type checking and conversion; see Peyton Jones et al. [2016] for the full explanation. This appearance of dynamic typing may appear worrying; but once again, our phase separation saves us. The dynamic checks during configuration processing will all happen at *config-time*, so that we have no dynamic typing in the more critical run-time.

A.2 Reinterpretation of effects

Here, we will see how one effect can be interpreted in terms of another, taking advantage of the full generality of *handle'* (Fig. 7). The *handle'* function is the only place in the Kern-mantle architecture where the order of the strands in the mantle matters. Indeed, the tasks and the workflow are implemented independently of the ordering of the effects, as they use the *InMantle* constraint instead of an explicit ordering. Effectively, *Rope* considers the mantle to be a *set*, not a list. Each *handle* call offers an interpretation of top strand of the mantle and "pops" it, so the chain of *handle* calls fully interprets all the effects in the *mantle*.

To understand how we interpret one effect in terms of another, we start by inspecting the type of the first argument to *handle'*:

```
  -- Install an effect handler that may be dependent
  -- on other effects
handle' :: ∀ name eff core mantle a b.
           (  (∀ x y. Rope core ((name :– eff) : mantle) x y
                 →    core x y)
            → (∀ x y. eff x y → core x y))
         → Rope core ((name :– eff) : mantle) a b
         → Rope core mantle a b
handle' interp rope = Rope (λhandlers →
  let run :: ∀ x y. Rope core ((name :– eff) : mantle) x y
             → core x y
      run (Rope runner) = runner (HCons (interp run) handlers)
  in run rope)
```

Figure 7. Dependent effect handler installation

```
  (∀ x y. Rope core  (  (name :– eff) : mantle) x y
                    → core x y)
→ (∀ x y. eff x y → core x y)
```

Accordingly, in order to interpret *eff x y*, the interpretation function can either express it directly in terms of *core x y*, or the function can use the interpretation of the entire *Rope*, with all its strands, to process *eff x y*. This generality allows us to write an interpretation of one effect in terms of another one, further down the mantle—or even itself, if a finite fixpoint of interpretation exists.

We can use the examples of the previous sections to show an example of this reinterpretation. Both *ReadResource* and *GetOpt* were interpreted in terms of the *Config* core type, using similar logic. Instead of implementing that logic twice, we can implement it once for the interpretation of *GetOpt* and then interpret *ReadResource* in terms of *GetOpt*:

```
interpretReadResourceGeneric
  :: (Arrow core, InMantle "options" GetOpt mantle
     , InMantle "io" (Kleisli IO) mantle)
  ⇒ ReadResource a b
  → Rope core mantle a b
interpretReadResourceGeneric (ReadDataSet ident) =
  proc () → do
    actualPath ← strand @"options" opt ⊸ ()
    strand @"io" (Kleisli loadDataSet) ⊸ actualPath
  where
    opt = GetOpt ident
                 ("The source of dataset " ++ ident)
                 (Just ident)
interpretReadResourceGeneric (ReadRawFile ident) = ...
```

Now the interpretation of *ReadDataSet* is more apt, as it is abstract from the logic of storing and retrieving option values. We derive the option name, the help text and the default physical path from the data set identifier. This logic could work over different *core* types, so it is easier to reuse. Taking our *workflow* from previous section, we can interpret all its mantle this way:

```
workflow :: (Arrow core
           , InMantle "resources" ReadResource mantle)
         ⇒ Rope core mantle () DataSet

interpretGetOpt :: Arrow eff
                ⇒ GetOpt a b
                → (Config String Dynamic ↠ eff) a b
interpretIO :: Applicative f
            ⇒ Kleisli IO a b
            → (f ↠ Kleisli IO) a b

interpretedWorkflow ::
  (Config String Dynamic ↠ Kleisli IO) () DataSet
interpretedWorkflow =     strip $
    -- "io" strand directly contains Kleisli IO effects:
  handle @"io"       interpretIO       $
  handle @"options" interpretGetOpt $
  handle' @"resources" (λinterp eff →
    interp (interpretReadResourceGeneric eff)) $
  workflow
```

The interpretation function we give to the *handle* `@"resources"` call is fed a function that can turn a *Rope* back into its core. This rope can make use of the `"resources"` strand or of any strand that has still not been woven in the *core*. So here, given that `"io"` and `"options"` strands will be woven afterwards, they are usable by *interpretReadResourceGeneric*.

B Interpretation function in biomodeling case study

```
interpretBiomodel
    :: (InMantle "options" GetOpt mantle
      , InMantle "logger" LogEff mantle
      , InMantle "resources" AccessResource mantle
      , Arrow core)
    ⇒ Biomodel a b
    → Rope core mantle a b
interpretBiomodel model = proc params → do
    -- Load the initial conditions:
  ics  ← getOpt "ics"
                ("Initial conditions: "
                  ++ show (odeVarNames model))
                (Just $ odeInitConds model) ⊸ ()
    -- Load simulation parameters:
  start  ← getOpt "start" "T0 of simulation"
                  (Just 0) ⊸ ()
  end    ← getOpt "end" "Tmax of simulation"
                  (Just 50) ⊸ ()
```

```
  points ← getOpt "timepoints"
                  "Num timepoints of simulation"
                  (Just 1000) ⤙ ()
    -- Simulate the system:
  strand @"logger" LogEff ⤙ "Start solving"
  let timeVec =
        toVector computeSolutionTimes start end points
      ! resMtx =
        runModel (odeSystem model params) ics timeVec
        -- The ! is to ensure that logging
        -- completion happens after computation
  strand @"logger" LogEff ⤙ "Done solving"
  strand @"resources" (WriteResource "res" "csv") ⤙
    serializeResultMatrix (odeVarNames model)
                          timeVec resMtx
  viz ← generateVisualization ⤙
    (timeVec, odeVarNames model, resMtx)
  strand @"resources" $ WriteResource "viz" "html" ⤙
    serializeVisualization viz
  id ⤙ odePostProcess model timeVec resMtx
  where getOpt n d v =
        strand @"options" (GetOpt n d v)
```

C Our implementation

The implementation is available as a tarball and packaged as the `kernmantle-0.1.0.0` cabal library. A stack configuration is also provided.

Control structures. In our presentation of *Arrow*, we limited ourselves to the *arr* and *first* methods for the sake of simplicity since they are the only methods used by the desugaring of the Arrow notation. Notably, we did not mention $(***)$ which is by default implemented in terms of *first* and $(\circ)$. As *Rope* just relies on the implementation of $(***)$ of the *core* and propagates it, one could use instead of *Kleisli* an arrow that always parallelizes tasks composed with $(***)$ in order to obtain a workflow with parallel execution, though it requires the tasks to be explicitly composed with $(***)$. Besides *Arrow*, we implemented a range of class instances for *Cayley* and *Rope*, including *ArrowChoice*, *ArrowLoop*, *ArrowZero* and *ArrowPlus*. This means that a workflow expressed with *Rope* can go beyond a pure acyclic directed graph, and contain conditional branching, cycles and alternatives (for instance to do error recovery). Additionally, as we mentioned it other underlying control structures could be used instead of *Arrow*, so we provide instances for all relevant classes in the profunctors package (*Profunctor*, *Strong*, *Choice*, etc.) which can be considered superclasses of the *Arrow* stack of classes. We also provide a *Traversing* instance, which allows to traverse a structure with any *Rope* (that has a *Traversing* core, like *Cayley* and *Kleisli* provide) with any *Traversal* (from the lens package). Few of these classes need to be implemented by hand, as many can be derived from the instances of *Cayley*, so we inherit the compliance with *Arrow*, *Category* and *Profunctor* laws. We make heavy use of the *DerivingVia* extension to GHC [Blöndal et al. 2018].

Naming. Some names may differ slightly from those presented in the paper. The *Handlers* are called *Weavers*, and *handle* is called *weave*.

Record of handlers. We use the vinyl package to implement our record of handlers. This allows us to be parametric over the type of record used. We frequently store the handlers in a compact array rather than a linked list, which means the *strand* function has $O(1)$ access to the handler it requires. We just transform this array into a linked list when we want to call *handle*.

Rope type parameters. The order of the *core* and *mantle* parameters in *Rope* type is flipped in the implementation, and we often use constraint aliases to simplify the uses of *InMantle*. *InMantle* is also called *InRope* as it needs to be parameterized over the full rope (due to particulars of vinyl's interface). As described above, *Rope* is polymorphic over the record of handlers it uses, so *Rope* has an extra type parameter *record*.

Strand names. We use GHC's *OverloadedLabels*[12] extension instead of type applications, so that *handle* and *strand* explicitly take an extra parameter. This means that forgetting to name the effect we want to handle or execute will result in an error at compile-time due to a missing function parameter.

Config. We did not need to implement our *Config* type. For specific configuration sources, applicative functors already exist that have the exact same behavior as *Config*. For instance, if one is only interested—like we were when implementing our use case with Kernmantle—in configuration through command-line arguments, then the *Parser* applicative functor from optparse-applicative provides everything needed, as it morally is already a combination of a writer (accumulating options, help text and default values) and a reader (constructing a result based on the actual configuration that was parsed). This shows that this pattern integrates quite nicely with a pre-existing ecosystem.

Caching. We implement the caching store thanks to cas-hashable and cas-store, which provide a *cacheKleisliIO* function that can back up with the store any $a \longrightarrow IO\ b$ function with the right constraints on *a* and *b*. The store package is used to serialize results.

Biochemical modeling. We use the hvega (for visualization) and hmatrix-sundials (for ODE solving) packages in the implementation of our biochemical modeling use case.

[12] https://ghc.readthedocs.io/en/8.0.1/glasgow_exts.html#overloaded-labels

D Distributed execution of a workflow

One possible direction of future work in Kernmantle is to extend it to work in a cloud environment, where the same workflow is sent to several workers simultaneously, each of them executing only part of the workflow. In the rest of the subsection we speculate on how this extension can be built, calling it Distributed Kernmantle (DKM). This illustrates that the abstractions we have described support non-trivial extensions.

We will require the same type of store, hashing, serializing, and naming scheme as presented in section 5.2, except that this time we will need to make the store distributed, by using, for example, an Amazon S3 bucket or an HDFS filesystem as the store. Given a cluster of machines, we divide a DKM instance running on it into three layers:

- The scheduling layer
- The job layer
- The distributed store

In DKM, a job is a Kernmantle executable: a binary that runs a workflow. The job layer is made of several workers, which are machines with no prerequisites other than being able to run a statically linked executable. This executable runs a Kernmantle workflow that will make use of some primitive effects that will be interpreted against the store to determine which worker will perform which tasks in the workflow. The important point is that every worker that takes part of a job runs the exact same executable as the other workers, and therefore has knowledge of the same full workflow. This means that, for example, the automatic naming feature of the cache can derive the same task identifiers on each machine and that two workers can identify the same sub-task in the same way. This method of identifying tasks is conservative: if the same shared computation happens twice in two different positions in the workflow, it will be given a different identifier, and therefore will not be shared between those two positions. This is why the user should be able to override a task's identifier when they want to enforce sharing between these two positions—something that our *withStore* and *caching* methods already take care of.

In addition to a task identifier, a task still needs to hash its inputs to identify a deterministic computation to perform. Not all tasks in a workflow need to have hashable inputs, but those that do not cannot be distributed across the cluster and will not be cached. A task which is identified *and* has hashable inputs and serializable outputs is called a *shared task*. Lightweight pure functions as well as non-deterministic functions should not be shared, and each worker will recompute them instead of sharing them.

When a worker encounters a shared task T in a workflow, it queries the store: if a lock file indexed by the task's hash H is already present, it will look for a different task to perform. Assuming that a cached result for T is not already in the store (but skipping it otherwise), the worker places a lock file under H and starts the task. When it is done, it writes the result and continues. While the worker computes, it re-locks H at regular intervals. Lock files are temporary—they automatically expire after a certain amount of time—so a dead worker does not hog a computation. This is why workers should take care of maintaining the locks while they compute. All of this means that the only communication point between the workers is the store. Apart from it, they are decentralized, and do not have to know one another.

In order to support a fully distributed system, we also use a distributed scheduler. Each worker would host a daemon process, looking for new tasks to start. This daemon would hook into the distributed store used above, but would otherwise not interact with Kernmantle.

To submit new jobs, we need a scheduling layer. It is also decentralized. The user will push a new job by uploading it in a given place in the store. On the workers, a DKM daemon runs that checks submissions of new jobs in the store. Each job is attached a priority, and depending on its current load and on the new job's priority it will download the executable and start running it or not. Determining whether a worker should run a given job can be entirely stochastic: if we grade a job's priority between 0 (no worker should care) and 1 (super important), and if we qualify the worker's availability also between 0 (completely busy) and 1 (doing nothing), then by multiplying the two we get the probability that the worker should start the job. With enough workers, doing so will have the same effect than fixing the number of workers that should run the job, on average, albeit with fewer synchronization needed between them. But, if needed, that synchronization would be easy to add, by using the same arrow framework that the workers use: starting a job on a given worker would be a task like any other, we would simply repeat that task (running an executable) over the required number of times N we want it to run throughout the cluster, and that task would be indexed by the executable SHA1, by an job-identifier (chosen by the user) and a number from 1 to N. The daemon would just need one extra primitive: watching a folder (in the store) for new submissions.