Composing Algebraic Effects into Tasks and Workflows

YVES PARÈS, Tweag I/O
JEAN-PHILIPPE BERNARDY, University of Gothenburg, Sweden, Sweden
RICHARD A. EISENBERG, Bryn Mawr College, USA and Tweag I/O, UK

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 presents an approach toward combining algebraic effects with arrows (which we rebrand as *tasks*). This novel combination of techniques leads to a composable, modular way to design workflows such that they 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 will be applicable to new data science and machine learning applications and frameworks.

1 INTRODUCTION

Many data science or machine learning applications and architectures follow a predictable pattern: a succession of computation *tasks*, each one receiving a set of inputs and computing a set of outputs. While the ubiquitous term *pipeline* is evocative of a linear flow, as would befit a fluid in a physical pipeline, this term does not depict the full generality of how computations are typically linked. Data does not necessarily flow *linearly*, but rather it flows in a directed and (often) acyclic graph (DAG) of tasks. Pipelines are also called *workflows* in the data science community.

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 a line or a block of lines in a dataset (Apache Spark [Zaharia et al. 2010], Apache Storm[1], or Apache 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]).

Accordingly, at any of these levels, we are very 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 and can be analyzed before any data is provided.

Our 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 has existed for some time: these are the *arrows* 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].

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

Authors' addresses: Yves ParèsTweag I/O, yves.pares@gmail.com; Jean-Philippe Bernardy, Department of Philosophy, Linguistics and and Theory of Science, University of Gothenburg, Sweden, Box 100, Göteborg, SE-405 30, Sweden, jean-philippe.bernardy@gu.se; Richard A. Eisenberg, Department of Computer Science, Bryn Mawr College, 101 N. Merion Ave., Bryn Mawr, PA, USA , Tweag I/O, UK, rae@richarde.dev.

2020. 2475-1421/2020/3-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Despite matching the requirements for workflows well, arrows have not, to our knowledge, been applied to fit 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 initially 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 many sequels), effect composition has not correspondingly been considered deeply with respect to arrows.

In this paper, we propose to handle effects in arrows via a novel application of the technique of *algebraic effects* [Plotkin and Pretnar 2013]. We offer the following contributions:

- Our new approach, *Kernmantle*, is the main contribution of this work. This approach allows us to meet the design goals outlined above: it is a framework for statically analyzable composable tasks, supporting algebraic effects. (Section 4)
- In explaining Kernmantle, we also refresh the presentation of arrows, rebranding them as *tasks*. We find this new terminology easier to work with and more evocative than the more technical *arrow*. We believe that arrows—hence, *tasks*—are underutilized, and believe our consolidation of this existing idea to be an independent contribution of this paper. (Sections 2.2–3)
- 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 these examples serve as inspiration to other practitioners to use these techniques to build composable workflows. (Section 5)

Beyond these main contributions, our paper includes a comparison to related work (Section 6) and some thoughts on future directions (Section 7)—including how Kernmantle could be used to structure distributed workflows (Section 7.3).

The code in this paper is presented in Haskell, and we provide executable sources (including parts of this paper that are elided in the text) as supplementary material, along with a library[3] we built using the architecture described here. Our choice of Haskell is well-motivated, due to Haskell's purity and support for arrow-notation [Paterson 2001], but inessential. The core idea of this paper is not about Haskell, but instead about a new way of structuring composable tasks. We recommend viewing the code merely as a formal *specification* of our ideas, not as the concrete *implementation*. Indeed, our real concrete implementation differs from the presentation here, which is intended more for exposition and specification than execution. We review the differences in Section 7.1.

2 BACKGROUND

2.1 Phases

Programmers in compiled languages are accustomed to thinking about two *phases* in which their program lives: 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.
- 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*.

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

- 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 by the programmer, while run-time execution is done by the client.

We introduce a third phase: *load-time*. It happens between compile-time and run-time: after the OS has loaded the application in memory but before the pipeline has started to process any data. These applications are started by domain experts, familiar with the internal workings of the application logic. After the application is started, it may take days to process data and return a useful result. Even though load-time can be considered to be the beginning of the run-time phase, separating it out 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 load-time can be interpreted and fixed. For example, 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 load-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 load-time analysis can *optimize* the pipeline. The computation associated with this optimization is performed *once*, in advance, instead of concurrently with data processing.

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

Note that load-time typically has access to more information than compile-time. Avoiding to encode more information in types, the API can be simpler. For example, loading might access configuration files or otherwise inspect the deployment environment, which may be different from the development environment used for compiling.

2.2 Tasks

The use of monads is a popular to model effectful computations, but computations constructed thus are not amenable to load-time analysis of their structure. They simply do not allow us to make the distinction between load-time and run-time. Tackling this limitation is the essential topic of this paper. To understand what we mean, consider the type of the monadic bind:

$$(\gg\!=) :: \mathit{Monad}\ m \Rightarrow m\ a \to (a \to m\ b) \to m\ b$$

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 know anything about the final result. This means that, when effectful computations are structured as monads, the computational effects are intertwined with each other (and to the final result): it is impossible to extract one kind of effect without extracting all effects and the associated (pure) resulting value[4].

Hughes [2000] proposes to instead structure effectful computations around *arrows*, which, in our view, correspond neatly to the notion of a task in a data pipeline.

In a nutshell, Hughes' 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. In the rest of this paper, we call such types of two arguments (be they tasks or not) *binary effects*, in contrast

[4]In certain cases one can use lazy evaluation to partially evaluate the computation and get only a subset of the effects—but achieving this relies on the global dynamic semantics of the program, and is only achievable in certain cases.

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

Though we present our work in Haskell, we stress that the general idea of composable tasks is independent of the features of that particular language. Naturally, the precise encoding of these ideas in other languages will differ.

We have found the standard treatment of arrows in Haskell (as embodied in the *Control.Arrow* module shipped with GHC) to be challenging to pick up. We thus propose to repackage and rename this idea, describing computations in terms of *tasks*. Despite the fact that Kernmantle is implemented using full arrows, we present its essence here in terms of tasks; the relationship between our presentation in this report and the actual implementation is discussed in Section 7.1.

A task *T* supports the following operations:

```
class Task t where
  id    :: t a a                  -- Identity
  (∘)   :: t b c → t a b → t a c  -- Composition
  arr   :: (a → b) → t a b        -- Pure task
  first :: t a b → t (a, c) (b, c) -- Passing extra data along a Task for later use
```

These operations will be the building blocks we need to construct our tasks and pipelines.

2.3 Haskell's arrow notation

In some examples that follow, we use GHC/Haskell's `-XArrows` syntactic extension [Paterson 2001], which we describe briefly here. Readers familiar with this extension can safely skip this subsection.

We will give a high-level intuition based on example; the original paper has a more formal treatment. 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 *Task* 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 consume 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 $\lambda x \rightarrow (x, x)$ 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 we see in the explicit form above. This allows us to concentrate on the flow of data, not wrangling combinators; in much the same way that lambda-binders are easier to manipulate than SKI 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 $\leftarrow$ are *not* in scope between the $\leftarrow$ and the $\prec$. 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, load-time analysis.

3 BASIC TASKS

Before we present Kernmantle, we show how the *Task* structure can be used to model simple effects.

3.1 Pure Functions

Pure functions can be used as tasks[5]:

```
instance Task (→) where ...
```

3.2 Kleisli

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

```
newtype Kleisli m a b = Kleisli (a → m b)
runKleisli :: Kleisli m a b → a → m b
runKleisli (Kleisli action) = action

instance Monad m ⇒ Task (Kleisli m) where ...
```

But wait: we have said that monads were trouble, because they cannot be analyzed at load-time. Accordingly, a good pipeline 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* task 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 fact that any monad can be lifted to the *Kleisli* 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* task allows an effect of writing information into a load-time channel—as long as that information does not depend on the input data[6]. This could be useful for, e.g., *load-time* logging.

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

[5]We suppress less interesting parts of our implementation in this text. The full, self-contained code is available as supplementary material.

[6]We use an *A* suffix to distinguish tasks from monads for two reasons: *A* is inspired by the word *arrow*, and *T* (for *task*) is already taken by monad transformers.

This *WriterA* builds upon an underlying task type *task* (much like monad transformers do). If we specialize this underlying task 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* task 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 load-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 task:

```
instance (Monoid w, Task arr) ⇒ Task (WriterA w arr) where ...
```

Given an existing task *t* :: *task a b*, we can make it write at a side result of *x* :: *w* at load-time by writing *WriterA x t* :: *WriterA w task 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, Task t) ⇒ Task (Cayley f t) where ...
```

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. Because applicative functors do strictly less than monads, every monad is also an applicative functor. (Accordingly, in Haskell, the *Applicative* class is implied by the *Monad* class.) Interestingly, *Cayley* gives us exactly what we need to implement the *WriterA* task in terms of the *Writer* applicative functor—without losing the ability for load-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*) (→) *a b* ≈ *Writer w* (*a* → *b*) ≈ (*w*, *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 the *Cayley* type can be used repeatedly to create several layers on top of a task, we are going to use a type operator as an alias to enhance readability in the rest of the paper:

```
type (↠) = Cayley
infixr 1 ↠
```

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

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

```
type BinEff = ⋆ → ⋆ → ⋆   -- The kind of our binary effects
  -- A Strand names a binary effect; Symbol is just a compile-time ("type-level") string
data Strand = Symbol :– BinEff
  -- A Mantle is just a list of strands
type Mantle = [Strand]
  -- Extract the effect from a Strand
type family StrandEff (t :: Strand) :: BinEff where StrandEff (_ :– eff) = eff
  -- A Weaver is an interpreter for the effect in strand, interpreted in the core task type core
type Weaver (core :: BinEff) (strand :: Strand) = ∀ x y. StrandEff strand x y → core x y
  -- Weavers is a heterogeneous list of Weavers, including handlers for many Strands
data Weavers (core :: BinEff) (mantle :: Mantle) where
  WNil  :: Weavers core '[]
  WCons :: Weaver core strand → Weavers core mantle → Weavers core (strand : mantle)
  -- A Rope core mantle is a task that can be executed in a core task type core
  -- when given interpretations for the effects in mantle.
newtype Rope (core :: BinEff) (mantle :: Mantle) a b = Rope (Weavers core mantle → core a b)
instance Task core ⇒ Task (Rope core mantle) where ...
```

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

Even though we use an arrow-like symbol here, *Cayley* is definitely not function-like. For instance, layers cannot be reordered: $(Reader\ r \twoheadrightarrow Writer\ w \twoheadrightarrow eff)\ a\ b \equiv r \to (w, eff\ a\ b)$ is not isomorphic to $(Writer\ w \twoheadrightarrow Reader\ r \twoheadrightarrow eff)\ a\ b \equiv (w, r \to eff\ a\ b)$. The type $f \twoheadrightarrow g$ 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 a task. 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* (*SomeTask 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 pipeline. Also, given that every *Monad* is also an applicative functor, it means any monad library could be used to build such computations.

4 THE KERNMANTLE FRAMEWORK

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

Essentially, Kernmantle is the composition of two well-studied techniques from the functional programming world: arrows (which we are calling tasks), and algebraic effects. We have seen a thorough introduction to tasks, but let us now focus on algebraic effects. The key idea behind an algebraic-effects 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—we dive into Kernmantle.

```
data LogEff a b where     -- A binary effect type encoded as a GADT
  LogEff :: LogEff String ()
task1 :: Task t ⇒ t X Y
task2 :: Task t ⇒ t Y Z

  -- A pipeline using this effect:
pipeline :: (Task core, InMantle "logger" LogEff mantle)
         ⇒ Rope core mantle X Z
pipeline = 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
```

Fig. 2. A pipeline with a logging effect

Ropes. The main Kernmantle datatype, *Rope*, is presented in Fig. 1. We call the target type of our interpretation functions the *core*. Each effect type to be interpreted in the *core* is given a name. A named binary effect is called a *Strand*, as our rope will "weave" together several strands of effects. The collection of all the strands of effects is called the *mantle* of the rope. The interpretation function of a strand is called a *Weaver*, and all the weavers are collected in a *Weavers* heterogeneous list.

In a library built using the Kernmantle architecture, each task exposed to the user is therefore implemented as a *Rope*. The *core* provides the fundamental operations that allow you to build a pipeline, and the *mantle* provides the effects our tasks will want to execute, hence the name "kernmantle" (core-mantle)[7].

Example pipeline. 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[8]. The logging effect is encoded with the *LogEff* type, and we provide a interpretation function *interpretLogEff* separately.

The pipeline uses this effect under the name `"logger"` and uses the *strand* function (see figure 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 (figure 3) asserts this requirement and provides a method *findWeaver* 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

[7]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.

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

```
  -- InMantle name strandEff mantle says that effect strandEff is named name in the mantle
class InMantle (name :: Symbol) (strandEff :: BinEff) (mantle :: Mantle) where
    -- Extract an interpretation from strandEff to a core from a Weavers
  findWeaver :: Weavers core mantle → strandEff x y → core x y

  -- These two instances effectively look up a name in mantle.
  -- The overlap is harmless when names are unique
instance {-# overlapping #-} InMantle name strandEff ((name :- strandEff) : mantle) where
  findWeaver (WCons w _) eff = w eff
instance InMantle name strandEff mantle ⇒ InMantle name strandEff (other : mantle) where
  findWeaver (WCons _ ws) eff = findWeaver @name @strandEff @mantle ws eff

  -- 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 (λweavers → findWeaver @strandName @strandEff weavers eff)
```

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

with different names. We also note that this pipeline places a requirement on the *core*: that it should be an *Task*, so the whole *Rope* can be a task too. Our use of arrow-notation desugars to use the combinators we defined as part of the *Task* class[9].

Running the pipeline. A *Rope* is just a function closure, as we can see in the definition of the *Rope* type. It takes a *Weavers*—a heterogeneous list of effect handlers—to a result in the core effect type, *core*. Accordingly, the interesting data stored in a *Rope* is closed over by the function stored in the *Rope*; it is this function which selects out the correct effect handlers and sends the desired effects to them.

To run a *Rope* and execute its effects, we call the *weave* function in Fig. 4. The easiest way to understand *weave* is to look at its simpler form, *weave'*—specifically its type. We see there that *weave'* takes a way to interpret *strandEff* (the effect supported by the strand in question) in the *core* effect. In our running example here, we will 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 *weave'* (and *weave*) has a higher-rank type [Peyton Jones et al. 2007]. Given the interpretation function, *weave'* allows us to treat a *Rope* with the *strandEff* in its mantle as one without support for that effect. We can see how nested calls to *weave'* can thus handle multiple effects. The core effect from the bare, mantle-less *Rope* can then be executed with a call to *untwine*.

The more general *weave* function is just like *weave'*, except that the interpretation function takes an extra argument: an interpretation function for the *Rope* with its *mantle* intact. In this way, outer layers of a mantle may use effects in inner layers, because the interpretation for the outer effect can access the interpretation of all of the *Rope*'s effects. The *weave'* simplification just ignores this possibility.

Here is how we actually execute the pipeline:

[9]We use the GHC extension `-XRebindableSyntax` so that the desugaring uses our *Task* combinators instead of the built-in *Arrow* ones.

```
  -- By supplying an effect handler, reduce the number of effects in a Rope's mantle
weave :: ∀ name strandEff core mantle a b.
        (   (∀ x y. Rope core ((name :– strandEff) : mantle) x y → core x y)
         → (∀ x y. strandEff x y → core x y))
      → Rope core ((name :– strandEff) : mantle) a b
      → Rope core mantle a b
weave interpFn rope = Rope (λweavers →
  let run :: ∀ x y. Rope core ((name :– strandEff) : mantle) x y → core x y
      run (Rope runRope) = runRope (WCons (interpFn run) weavers)
  in run rope)

  -- Simpler form of weave that does not allow dependent effects
weave' :: ∀ name strandEff core mantle a b.
          (∀ x y. strandEff x y → core x y)
       → Rope core ((name :– strandEff) : mantle) a b
       → Rope core mantle a b
weave' interpFn = weave (λ_ → interpFn)

  -- Take a Rope without a mantle and extract its core effect.
untwine :: Rope core '[] a b → core a b
untwine (Rope runRope) = runRope WNil
```

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

```
runPipeline :: X → IO Z
runPipeline x = let pipelineWithoutEffs = weave' @"logger" interpretLogEff pipeline
                in case untwine pipelineWithoutEffs 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*]):

```
interpretLogEffWriter :: LogEff a b → Kleisli (Writer [String]) a b
interpretLogEffWriter LogEff = Kleisli (λlogLine → Writer [logLine] ())
runPipelineWithWriter :: X → ([String], Z)
runPipelineWithWriter x =
  let pipelineWithoutEffs = weave' @"logger" interpretLogEffWriter pipeline
  in case untwine pipelineWithoutEffs 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.

5 PRACTICAL EXAMPLES WITH KERNMANTLE

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

5.1 A pipeline with abstract data sources

Here we present a *ReadResource* effect that gives a task the capacity to open an external resource, inspectable at load 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 sort of resources, either as structured data—whose type is known in advance—or as raw bytestrings.

Setting up our effect this way imposes a constraint: the resource identifier must be knowable *at load-time*. Because the choice of *ResourceId* is required in order to construct the binary effect, it will not have access to run-time data; recall that run-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 pipeline that accesses *ResourceId*s `"dataset1"` and `"dataset2"`:

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

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

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

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

Thus, when we interpret this pipeline, we need to instantiate *core* with a type that can be split into two parts: one relevant at load-time and one relevant at run-time. At this point, a concrete illustration is more helpful than an abstract description; here is the desired type of the interpreted pipeline:

```
interpretedPipeline ::
  (Set ResourceId                          -- Requirements collected at load-time
  , Map ResourceId FilePath → IO DataSet)  -- Run-time function taking the final mappings
```

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 pipeline has two separate components. If we wish to do only a load-time analysis, not to run the pipeline, we can just project out the first component of the tuple. If we want to run the pipeline, however, we use the first load-time component to set up the argument to the second run-time one. 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 is a task type—and the type above does not appear to be a task. Yet it is isomorphic to the task type (*Writer* (*Set ResourceId*) ↠

Reader (*Map ResourceId FilePath*) ↠ *Kleisli IO*) () *DataSet*; here, we are using the suggestive ↠ notation instead of *Cayley*. Here is the interpretation function:

```
interpretedPipeline = case untwine (weave' @"resources" interpretReadResource pipeline2) 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) = ...
```

Interpreting *pipeline* this way provides us the separation that we want between load-time and run-time. However, we can 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 load-time analysis of resources[10], 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 *interpretedPipeline* gets the following type:

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

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

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

If we wish to actually run the pipeline, 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 whether final configuration is complete and valid; and
(4) run the pipeline by feeding it the validated configuration.

The fact that we can extract the *ResourceId*s in advance of running the pipeline is crucial: this allows our program to examine its environment at load-time to determine whether some errors will happen at run-time. If the required resources are available, then the overall application can be allowed to proceed. 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.

[10] In fact, it is common enough so it is the very foundation of the related porcupine library.

5.2 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        -- Allow printing 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 intrepreted in the same way than *ReadResource*, using the same logic as *interpretReadResource*, by generalizing a little bit our *core* type:

```
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 *load-time*, so that we have no dynamic typing in the more critical run-time.

5.3 Reinterpretation of effects

The effects previously studied are all interpreted in the *core*, using the simplified *weave'*. Now, we will see how one effect can be interpreted in terms of another, taking advantage of the full generality of *weave* (Fig. 4). The *weave* function is the only place in the Kernmantle architecture where the order of the strands in the mantle matters. Indeed, the tasks and the pipeline 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 *weave* call offers an interpretation of top strand of the mantle and "pops" it, so the chain of *weave* 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 *weave*:

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

Accordingly, in order to interpret *strandEff 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 *strandEff 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
  :: (Task 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"
    (GetOpt ident ("The source of dataset " ⧺ ident) (Just ident)) ⤙ ()
  strand @"io" (Kleisli loadDataSet) ⤙ actualPath
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 *pipeline* from previous section, we can interpret all its mantle this way:

```
pipeline :: (Task core, InMantle "resources" ReadResource mantle)
         ⇒ Rope core mantle () DataSet

interpretGetOpt :: Task eff ⇒ GetOpt a b ⟶ (Config String Dynamic ↠ eff) a b
interpretIO     :: Applicative eff ⇒ Kleisli IO a b ⟶ (eff ↠ Kleisli IO) a b

interpretedPipeline :: (Config String Dynamic ↠ Kleisli IO) () DataSet
interpretedPipeline = untwine $
  weave' @"io"        interpretIO    $   -- "io" strand directly contains Kleisli IO effects
  weave' @"options"   interpretGetOpt $
  weave @"resources" (λinterp eff ⟶ interp (interpretReadResourceGeneric eff))
  pipeline
```

The interpretation function we give to the *weave* `@"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*.

5.4 A pipeline of cacheable tasks

If a task is run twice with the same input, we want to be able to reuse cached results[11]. 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 pipeline, 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.

One might think that using a dedicated effect wrapping the part of the pipeline that we want to cache would be the way to go. Indeed, this could be a valid implementation and—as presented in

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

Section 5.3—*weave'* would allow us to recursively interpret this sub-pipeline, but here we want to present an even simpler one. 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*:

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

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 $ λweavers → withStore mbName (f weavers)
```

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 pipelines 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 pipeline, 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 pipeline, 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 pipeline. 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 pipeline. This DAG can be observed at load-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.

5.5 Case study: a pipeline for computational biology

After seeing several effects used separately, let us use them in a single pipeline 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 pipeline solution must satisfy, then we will see how the Kernmantle framework can meet these needs, using the effects we have already seen.

A basic pipeline 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 (e.g., CSV)

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 pipeline 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 simply 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 pipelines.
(5) A pipeline of tasks should be platform-independent.
(6) A pipeline 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 5.4.
(2) Reading from configuration files is covered in Sections 5.1 and 5.2.
(3) Arbitrary functions can easily be encoded via $arr :: Task\ t \Rightarrow (a \to b) \to t\ a\ b$, 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 pipelines.

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* (depending on *AccessResource* and *GetOpt*)? 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 pipeline so that the *GetOpt* and *AccessResource* effects can use that

namespace to generate qualified option names and default paths. Just like for caching in section 5.4 we are going to use a class providing a method to wrap a *Rope* task:

```
type Namespace = [String]   -- components of a name, like "Model1.Proteins.ForwardSim"
class Namespaced eff where
  addToNamespace :: String → eff a b → eff a b
```

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

```
vanderpol :: Biomodel Double ()
vanderpol = ...
chemical :: Biomodel Double ()
chemical = ...
  -- no type signature: let type inference do the work
pipeline = proc () → do
  strand @"logger" LogEff ≺ "Beginning pipeline"
  addToNamespace "vanderpol"
    (proc () → do mu ← getOpt "mu" "mu parameter" (Just 2) ≺ ()
                  caching Nothing (strand @"bio" vanderpol) ≺ mu) ≺ ()
  addToNamespace "chemical"
    (proc () → do k ← getOpt "k" "Left-to-right reaction rate" (Just 2) ≺ ()
                  caching Nothing (strand @"bio" chemical) ≺ k) ≺ ()
  strand @"logger" LogEff ≺ "Pipeline finished"
  where getOpt n d v = strand @"options" (GetOpt 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 on disk the raw result, and a visualization of the running biochemical simulation. The interpretation function appears in Fig. 5. It is included as an illustration of the ease in composing these pieces—the details are not important.

We have one question left: what *core* should we use to interpret this *pipeline*? 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. 6.

A *main* action including the calls to *weave* to build such an effect is in our supplementary material.

6 RELATED WORK

6.1 Roots of *Rope*

Our inspiration for the *Rope* type comes from the algebraic effects literature, e.g., [Kiselyov and Ishii 2015; Plotkin and Pretnar 2013]. The *Rope* type is very close to a final encoding of a free task whose generator is a sum of several effects, as shown in Fig. 7.

```
interpretBiomodel
  :: (InMantle "options" GetOpt mantle
    , InMantle "logger" LogEff mantle
    , InMantle "files" AccessResource mantle
    , Task 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 @"files" (WriteResource "res" "csv") ⤙
    serializeResultMatrix (odeVarNames model) timeVec resMtx
  viz ← generateVisualization ⤙ (timeVec, odeVarNames model, resMtx)
  strand @"files" $ WriteResource "viz" "html" ⤙ serializeVisualization viz
  id ⤙ odePostProcess model timeVec resMtx
  where getOpt n d v = strand @"options" (GetOpt n d v)
```

Fig. 5. Interpreting the *Biomodel* effect

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

Fig. 6. The core effect for the biomodeling case study

```
type Rope' effects = FreeT (Sums effects)
newtype FreeT t a b = FreeT (∀ task. (Task task) ⇒ (∀ x y. t x y → task x y) → task 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
```

Fig. 7. Free task of sum of binary effects

The original idea of structuring effects around monads is due to Moggi [1991]. He showed that many types of computational effects can be structured in the form of monads. 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.

One way 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:

```
newtype FreeM f a =
  FreeM {runFree :: ∀ 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* $(f :+: g)$[12].

For reasons explained in our introduction, we choose instead to structure computation around tasks, and thus use support a *Task* constraint instead of *Monad*. Accordingly m is renamed to t and changed to take two parameters, obtaining the free task (*FreeT* in Fig. 7). Accordingly, the free task of a sum of effects (*Rope'*) is in fact very close to Kernmantle's *Rope*. The differences are as follows:

- 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.
- 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 *Weaver*, and the record itself is our *Weavers*.
- We do not universally quantify over the target task (arrow) but rather make it a parameter of *Rope* (where it is called *core*). This allows the programmer to conveniently set specific constraints on *core*, for example to enable caching or to replace *Task* by more lax constraints (*Category*, *Profunctor*, *Strong*) or more stringent ones (*ArrowPlus*, *ArrowChoice*, *Traversing*)[13]. If one so wishes, the *core* used by some task can even be set to a specific given type.

[12]The popular polysemy library is structured around a free monad in this way.

[13]*Category*, *ArrowPlus* and *ArrowChoice* are in the base package. *Profunctor*, *Strong* and *Traversing* are in the profunctors package. All of these classes are also implemented by *Cayley f eff* (when f is an *Applicative*). The only class that cannot

6.2 Comparison with existing frameworks for task graphs and data workflows

Kernmantle explores a new area of the workflow tool design space. It is not strictly a data analytics framework, like Apache Spark is. It is much more lightweight and does not provide any facilities or abstractions to collect data and spread a dataset over a cluster of workers, even though such features could be built on top of Kernmantle. Furthermore, it is agnostic about the analytics to carry out: Kernmantle does not feature dataset manipulation primitives or linear algebra types and operations. Indeed, choosing the right tasks is deliberately kept orthogonal to the pipeline structure, and is still up to a Kernmantle user. Kernmantle is also not a framework to describe a DAG of high-level, totally abstract, coarse-grained tasks, like Spotify Luigi or AirBnB Airflow is. Lastly, it is not a tool to describe how data flows between some sources, transformers, aggregators, and sinks, like Apache Storm, Beam or the pipes/conduit/streaming ecosystems in Haskell.

In contrast, Kernmantle is a *bare* pipelining framework, providing structure to applications or to libraries thanks to concepts that are rooted in formal computer science (as arrows are), and uses that structure to provide load-time facilities. Another important note is that the *Rope* type 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 make use of some function like *unsafeInterleaveIO*, or explicitly fork some threads.

7 DISCUSSION AND FUTURE WORK

This section presents our implementation of the Kernmantle framework, its current shortcomings and the work planned around it in the future.

7.1 Our implementation

All the work presented so far in this paper has been fully implemented[14]. Our implementation is close to the excerpts we showed, with the following adjustments.

Classes. We use the *Arrow* and *Category* typeclasses instead of *Task*, to be closer to the existing ecosystem. Though they are beyond the scope of this paper, we also implement a range of class instances for our *Cayley* and *Rope* types, including *ArrowChoice*, *ArrowZero* and *ArrowPlus*. Additionally, 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 are 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].

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

---

be propagated by *Cayley* and *Rope* is *ArrowApply*, which is not desirable because *ArrowApply* treats an arrow as *Monad*, creating all the problems we exposed in section 2.2.

[14] https://github.com/YPares/kernmantle/

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 specificities of vinyl's interface). As described above, *Rope* is polymorphic over the record of weavers it uses, so *Rope* has an extra type parameter *record*.

Strand names. We use GHC's *OverloadedLabels*[15] extension instead of type applications, so that *weave* and *strand* explicitly take an extra parameter. This means that forgetting to name the effect we want to weave 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[16], which provide a *cacheKleisliIO* function that can back up with the store any $a \to IO\ b$ function with the right constraints on *a* and *b*. The store package is used to serialize results.

We actually have two classes for cores that provide caching: *ProvidesCaching* and *ProvidesPosCaching* ("position-dependent" caching). The former works as presented in section 5.4. The latter is implemented only by cores able to automatically derive a task identifier from the position of the task in the pipeline, so the requirement on automatic identification of tasks—based on their position in the pipeline—is manifest at the type-level. We accordingly have two different functions for tasks: *caching* (requires a name and a *ProvidesCaching core*) and *caching'* (takes no name as it will find it via automatic identification, but therefore needs a *ProvidesPosCaching* core). For these reasons the *Maybe String* name parameter is absent from *withStore* in the implementation. This adjustment ensures us—at the type-level—that no cached task is left unnamed.

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

7.2 Current limitations of Kernmantle

Both in the design presented here and in our implementation (see 7.1), the utility of the framework hinges on finding the right *core* type for a use case. We showed in section 5.5 that this core type can (and should) be derived directly from the requirements of our application. In general, this core type need not be implemented by hand, but it can be composed out of *Kleisli* and *Cayley* layers. Nevertheless, designing the *core* is a step that requires some thinking and some care. We can mitigate this by remembering that, in a real-world application of Kernmantle, the person in charge of determining the effects (a developer), the person using these effects in a pipeline (a data scientist, analyst, machine learning engineer or modeler), and the person selecting the core and writing the interpretation functions (possibly another developer) do not need to be the same person. Therefore, they don't require the same set of skills. One of the main advantages of Kernmantle is this separation between these different concerns through the abstraction layers it introduces.

[15] https://ghc.readthedocs.io/en/8.0.1/glasgow_exts.html#overloaded-labels
[16] These libraries are currently distributed with funflow.

In the current formulation, stateful interpretation of effects (i.e., having interpretation functions that receive a state and modify it) is also possible only if the *core* type is able to carry that state. This could be alleviated by introducing some helper function that uses the same kind of extensible record trick we use for weavers to automatically manage some *State* monad *inside* the *Kleisli* layer of the core whenever a stateful weaver is needed.

Contrary to some existing workflow tools (like Luigi, Spark or Beam), Kernmantle does not allow subtasks of a pipeline to run over several workers. Haskell binary code is indeed much less portable than Python or Java bytecode. Previous work has been made to use Haskell in a distributed environment [Epstein et al. 2011], not by shipping actual code to be executed by workers but by shipping closures containing a reference (that is stable through the cluster as long as the workers are running the exact same executable) to the function to execute as well as its serialized inputs. The GHC *StaticPointers*[17] extension and distributed-closure and sparkle packages have been developed to address that need. However, we next discuss a plan to support distributed execution via Kernmantle without the need for distributed closures, by directly relying on the static nature of the pipeline itself.

7.3 Distributed execution of a pipeline

One possible direction of future work in Kernmantle is to extend it to work in a cloud environment, where the same pipeline is sent to several workers simultaneously, each of them executing only part of the pipeline. 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 5.4, 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 pipeline. 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 pipeline. 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.

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 simply 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

[17] https://ghc.readthedocs.io/en/8.0.1/glasgow_exts.html#static-pointers

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.

8 CONCLUSION

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

While Kernmantle is presented in terms of Haskell, we stress that the code in this paper is the *specification*, using Haskell as a formal language—it is not the implementation of a library.

ACKNOWLEDGMENTS

NovaDiscovery, Tweag colleagues, anonymous reviewers...

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 (Haskell 2018)*. Association for Computing Machinery, New York, NY, USA, 55–67. DOI: http://dx.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 (Haskell '11)*. Association for Computing Machinery, New York, NY, USA, 118–129. DOI: http://dx.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. DOI: http://dx.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. DOI: http://dx.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. DOI: http://dx.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). DOI: http://dx.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 (POPL '95)*. Association for Computing Machinery, New York, NY, USA, 333–343. DOI: http://dx.doi.org/10.1145/199448.199528

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

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.

Gordon D. Plotkin and Matija Pretnar. 2013. Handling Algebraic Effects. *Logical Methods in Computer Science* 9, 4 (2013). DOI: http://dx.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, and others. 2010. Spark: Cluster computing with working sets. *HotCloud* 10, 10-10 (2010), 95.