CircuitFlow: A Domain Specific Language for Dataflow Programming

Riley Evans, Samantha Frohlich[0000−0002−4423−6918], and Meng Wang[0000−0001−7780−630X]

University of Bristol, Bristol, United Kingdom

Abstract. Dataflow applications, such as machine learning algorithms, can run for days, making it desirable to have assurances that they will work correctly. Current tools are not good enough: too often the interactions between tasks are not type-safe, leading to undesirable run-time errors. This paper presents a new declarative Haskell Embedded DSL (eDSL) for dataflow programming: CircuitFlow. Defined as a Symmetric Monoidal Preorder (SMP) on data that models dependencies in the workflow, it has a strong mathematical basis, refocusing on how data flows through an application, resulting in a more expressive solution that not only catches errors statically, but also achieves competitive run-time performance. In our preliminary evaluation, CircuitFlow outperforms the industry-leading Luigi library of Spotify by scaling better with the number of inputs. The innovative creation of CircuitFlow is also of note, exemplifying how to create a modular eDSL whose semantics necessitates effects, and where storing complex type information for program correctness is paramount.

Keywords: eDSL, domain-specific languages, Haskell, Dataflow programming

1 Introduction

CircuitFlow's domain is *dataflow programming* [7], which deals with processing data through transformations with interlinking dependencies. Inputs are transformed into outputs by *tasks*, organised into *workflows* taking the form of Directed Acyclic Graphs (DAGs) encoding dependencies, where the directionality indicates the direction the data is flowing, and the acyclicity ensures that the data doesn't go round in circles. Dataflow programming is highly applicable with numerous uses spanning from scientific data analysis [10, 19] to machine learning [1, 41]. Examples include *Data Pipelines*, *CI Systems*, *Quartz Composer* [16] and *Spreadsheets*. It also has the following benefits:

Declarative Describing the shape of the DAG instead of just indicating the connections, provides a more user-friendly and declarative experience.

Implicit Parallelism Since each node in a dataflow is a pure function, it is possible to parallelise implicitly. The purity of the nodes means that outside of data dependencies encoded in the dataflow graph, no node can interact with another. Thus eliminating the ability for a deadlock to occur.

Visual The dataflow paradigm uses graphs. This provides the programs with a visual interpretation, allowing end-user programmer to reason visually about how data passes through the program, much easier than in an imperative approach [14].

Existing dataflow libraries such as Spotify's Luigi [32] or Apache's Airflow [2] have no mechanism to ensure the dependencies are valid. There is no static checking that the connections in the graph match up, which could cause runtime crashes, or even worse, the bug could go unnoticed and cause havoc in later tasks. Consider an example shown in the docs for Luigi [33] that is made up of two tasks: the first, `GenerateWords`, generates a list of words and saves it to a file; and the second, `CountLetters`, counts the number of letters in each of those words. An implementation of this in Luigi could have a very subtle bug! `GenerateWords` could write the words to a file separated by new lines, while `CountLetters` expects a comma-separated list. This shows a key flaw in this system, as it is up to the programmer to ensure that they write the outputs correctly, and then that they read that same file in the same way. This error, would not even cause a runtime error, instead, it will just produce the incorrect result. For a developer, this is extremely unhelpful: it means more time is used writing tests — something that no one enjoys. With good development practices, the risk is reduced, but as functional programmers, we know a better way: abstraction and static typing.

Why not eliminate all of this with an abstraction of the reading and writing of many different sources and types? The abstraction will help to ensure correctness of passing data via files by eliminating any possible duplicated code. Instead, just having a uniform interface to test. Then the abstract interface can be combined with the type system so that in each program, it is enforced that the types align.

This promotes the need for a new solution with such features that can safely compose tasks and make use of types to perform static analysis to ensure that dependencies are valid.

We present CircuitFlow, which takes a different line of attack from its predecessing plumbers like Luigi: rather than focus on how to compose tasks together, it defines a declarative language that describes how data flows through a workflow. In CircuitFlow, it would not be possible to feed the output of one task, with the type FileStore [String] into a task that expects a CommaSepFile [String]. The same example, written in CircuitFlow, is defined as:

```
generateWords :: Circuit '[Var] '[()] '[FileStore] '[[String]] N1
generateWords = functionTask (const ["apple", "banana", "grapefruit"])
countLetters :: Circuit '[CommaSepFile] '[[String]] '[FileStore] '[[String]] N1
countLetters = functionTask (map f)
  where
    f word = (concat [word, ":", show (length word)])
```

```
circuit :: Circuit '[Var] '[()] '[FileStore] '[[String]] N1
circuit = generateWords <+> countLetters
```

In this example, it will fail to compile, giving the error:

```
> Couldn't match type 'CommaSepFile' with 'FileStore'
```

Benefiting the user since the feedback loop of knowing if the program will succeed is reduced. Previously, the whole data pipeline had to be run, whereas now this information is available at compile-time.

Due to the type heft required for such a language, which includes DataKinds [39], Singletons [9], Type Families [31], Heterogeneous lists [18], Phantom Types (a brief introduction of which can be found in Appendix A of an extended version of this manuscript [36]), it will be embedded.

CircuitFlow draws its origins from monoidal resource theory [6], details of which can be found in Appendix C [36]. It is then compiled down to a Kahn Process Network (KPN) that executes the workflow in parallel, to provide the speed benefits of multi-core processors. The KPN used by CircuitFlow is capable of handling an exception in a task, without causing the full network to crash, allowing computation to continue after for successive inputs.

Contributions: A declarative eDSL for creating dataflow programs that:

- employs state of the art DSL design techniques, including indexed data types à la carte and principled recursion to provide interpretations for the AST.
- uses state of the art Haskell methods to produce a type-safe implementation.
- makes use of indexed functors, extended to support multiple indicies, to construct a type-indexed AST in conjunction with an indexed monadic catamorphism to provide a type-safe translation to a KPN.
- has a strong mathematical grounding in monadic resource theories providing confidence that the language can represent all dataflow diagrams.
- has appealing preliminary benchmark performance against another competing library — outperforming Luigi by almost 4x on large numbers of inputs.
- exemplifies how to create such a language in a modular manner.
- uses the first known implementation of a Kahn Process Network in Haskell.

Examples that demonstrate the language's applicability:

- Machine learning: preprocessing of real world song data in comparison to Spotify's Luigi.
- Build systems: the thesis this paper is based on was compiled using CircuitFlow (details in Appendix B [36]).

2 CircuitFlow Language

A use case for CircuitFlow is building data pipelines for machine learning. Consider the example where an audio streaming service would like to create a playlist full of new songs to listen to. This could require a machine learning model that

can predict songs based on the top ten artists and songs that the user has listened to over the last three months. However, each of the months' data is stored in different files that need aggregating together before they can be input into the model. This problem can be drawn up as a dataflow diagram like Figure 1. To achieve this preprocessing, a software developer at said audio streaming service would need to use the following key features of the CircuitFlow language.

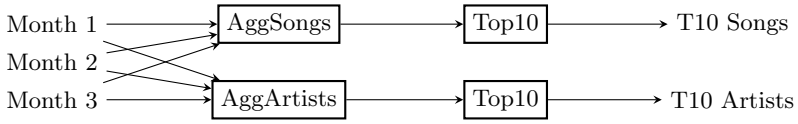


Fig. 1: A dataflow diagram for pre-processing the song data

2.1 DataStores

Dataflow programming revolves around transforming inputs into outputs. Thus the first thing the language needs is a way of getting inputs and writing outputs. For the preprocessing example, this corresponds to a way of interfacing with the different months of song data; a way to pass on the aggregated songs and artists to the top ten calculators; and finally somewhere to store the preprocessed output ready for the machine learning model. In CircuitFlow, DataStores are used to pass values between different tasks, in a closely controlled manner. To abstract over the different ways of storing data, they are defined as a type class:

```
class DataStore f a where
  fetch  :: f a → IO a
  save   :: f a → a → IO ()
  empty :: TaskUUID → JobUUID → IO (f a)
```

The type class provides a way of extracting a value from a DataStore (fetch), a way to write to one (save), and a way of creating an empty one for a specific task. Although the user can define their own, the library comes with predefined DataStores, the simplest is a Var, based on MVars (mutable locations).

```
newtype Var a = Var {unVar :: MVar a} deriving (Eq)
instance DataStore Var a where
  fetch = readMVar · unVar
  save = putMVar · unVar
  empty _ _ = Var <$> newEmptyMVar
```

Var doesn't use its id arguments in empty, however, other predefined stores, such as FileStore and CSVStore, use them to decide where to place the files created.

Combined DataStores A special case of a DataStore, they allow the interfacing with typed lists, not just a single type. The typed list is a variation on HLists: IHList (defined below). Combined data stores are automatically derived from existing DataStore instances, making it easier for tasks to fetch from multiple inputs by supplying fetch'. (Since tasks can only have one output, there is no need for a save' function.)

```
data IHList (fs :: [∗ → ∗]) (as :: [∗]) where
  HCons' :: f a → IHList fs as → IHList (f ': fs) (a ': as)
  HNil'  :: IHList '[] '[]
class DataStore' (fs :: [∗ → ∗]) (as :: [∗]) where
  fetch'  :: IHList fs as → IO (HList as)
  empty' :: TaskUUID → JobUUID → IO (IHList fs as)
```

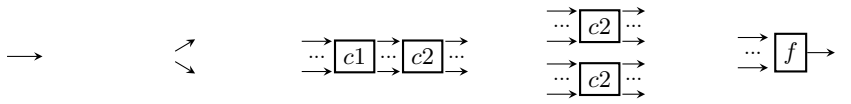
2.2 Circuit Type

A Circuit represents some computation that has some number of inputs and outputs. In order to statically check dependencies, the Circuit type needs to store a lot of information.

```
Circuit (insContainerTypes  :: [∗ → ∗]) (insTypes  :: [∗])
        (outsContainerTypes :: [∗ → ∗]) (outsTypes :: [∗]) (nIns :: Nat)
```

It has five type parameters: insContainerTypes, a type-list of storage types, for example '[VariableStore, CSVStore]; insTypes, a type-list of the types stored in the storage, for example '[Int, [(String, Float)]]; outsContainerTypes and outsTypes mirror that the examples above, but for the outputs instead. The container and value types are separate, due to the need for them to be "unapplied" for the DataStore typeclass. Unfortunately, GHC requires a little more information to perform this match check, such as the seemingly superfluous nIns, a type-level Nat that is the length of the input lists.

2.3 Circuit Constructors



(a) id (b) replicate (c) c1 <=> c2 (d) c1 <> c2 (e) task f

Above shows the core constructors of the language along with their diagrammatic representation. Here the relation to resource theories is apparent, the constructors in this library make up a SMP, establishing them as a resource theory able to represent any DAG. More details can be found in appendices C and

D [36]. The diagrammatic interpretation also makes translation from dataflow diagrams, such as Figure 1, to CircuitFlow code easy.

In the language, there are two types of constructors: those that create basic circuits and those that compose them. The behaviour of the constructor is recorded within the types. Here are the types of some basic circuits:

```
id        :: DataStore' '[f] '[a] ⇒ Circuit '[f] '[a] '[f]    '[a]    N1
replicate :: DataStore' '[f] '[a] ⇒ Circuit '[f] '[a] '[f, f] '[a, a] N1
```

Consider the id constructor, for convenience the nins parameter is shorted with type synonyms, e.g. N1∼'Succ 'Zero. It can be seen how the type information for this constructor states that it has 1 input value of type f a and it returns that same value. Each type parameter in id is a phantom type, since there are no values stored in the data type that use the type parameters. The replicate constructor states that a single input value of type f a should be input, and that value should then be duplicated and output. There is also a swap constructor that takes two values as input and swaps their order, and dropL / dropR constructors that will take two inputs and drop the left or the right one respectively.

To use these basic circuits, CircuitFlow provides two constructors named 'beside' and 'then' to compose circuits. The definition of these constructors will require type level calculations. This is where closed type families [8] come in, allowing for type level versions of (+) and (++) [18] (requiring PolyKinds [39]).

The 'Then' Constructor , denoted by <=>, is used run one circuit, *then* another, encapsulating the idea of dependencies. Through types, it enforces that the output of the first circuit is the same as the input to the second circuit.

```
(<=>) :: (DataStore' fs as, DataStore' gs bs, DataStore' hs cs)
  ⇒ Circuit fs as gs bs nfs → Circuit gs bs hs cs ngs → Circuit fs as hs cs nfs
```

It employs a similar logic to function composition $(\cdot) :: (a \to b) \to (b \to c) \to (a \to c)$. The resulting type from this constructor uses the input types from the first argument fs as, and the output types from the second argument hs cs. It then forces the constraint that the output type of the first argument and the input type of the second are the same — gs bs.

The 'Beside' Constructor , denoted by <> is used to run two circuits at the same time. The resulting Circuit has the types of the two circuits appended together.

```
(<>) :: (DataStore' fs as, DataStore' gs bs, DataStore' hs cs, DataStore' is ds)
  ⇒ Circuit fs as gs bs nfs → Circuit hs cs is ds nhs
  → Circuit (fs :++ hs) (as :++ cs) (gs :++ is) (bs :++ ds) (nfs :+nhs)
```

This constructor works by making use of the :++ type family to append the input and output type list of the left constructor to those of the right constructor. It also makes use of the :+ type family to sum the number of inputs.

Tasks are made using a smart constructor task, which requires a type level Length. To save boiler-plate, CircuitFlow also provides more handy task smart

constructors such as functionTask. This particular smart constructor allows a simple a → b function to be promoted to a task. It comes in useful returning to the music preprocessing example as it simplifies the definition of a task that finds the top ten songs or artists: functionTask (take 10).

2.4 CircuitFlow in Action

```
preProcPipeline = organiseIns <<< (       (aggSongs  <<< top10 "t10s.csv")
                                  <<< (aggArtists <<< top10 "t10a.csv"))
```

The above CircuitFlow circuit solves the music processing example. organiseIns replicates the input values so that they are passed into both aggSongs and aggArtists. Again, it can be seen how this structure of tasks directly correlates with the dataflow diagram previously seen in Figure 1. This helps to make it easier when designing circuits as it can be constructed visually level by level.

2.5 mapC operator

Currently a circuit has a static design: once created it cannot change. There are times when this could be a flaw in the language. For example, when there is a dynamic number of inputs. CircuitFlow's mapC allows for dynamic circuits. This constructor maps a circuit on an input containing a list of items. The input is fed one at a time into the inner circuit, accumulated back into a list, and then output.

```
mapC :: (DataStore' '[f] '[[a]], DataStore g [b])
   ⇒ Circuit '[Var] '[a] '[Var] '[b] N1 → Circuit '[f] '[[a]] '[g] '[[b]] N1
```

3 CircuitFlow Under the Hood

This section explores the embedding of the CircuitFlow language into Haskell and how it is translated down to be executed.

3.1 Circuit API

The constructors for the language are actually *smart constructors* [34], providing a more elegant way to build the AST that represents the circuit. They bring the benefits of extensibility and modularity usually found in a shallow embedding, while still having a fixed core AST that can be used for interpretation.

IFunctor The fixed core AST is implemented via a jacked up version of the traditional capturing of an abstract datatype as a fixed Functor story [12]. Instead of Functor, a type class called IFunctor [25] (also known as HFunctor [17]) is used as it is able to maintain the type indices, which in the case of CircuitFlow, are the all important dependency phantom type parameters. IFunctor can be thought of as a Functor transformer: it is able to change the structure of a Functor, whilst preserving the values inside it. IFunctors can also be used to mark recursive points of data types, as long as they are paired with a matching IFix to tie the recursive knot. As Circuit has five type parameters, it needs $\mathsf{IFunctor}_5$ and IFix_5.

```
type (⇝) f g = ∀a.f a → g a
class IFunctor iF where
  imap :: (f ⇝ g) → iF f ⇝ iF g
newtype IFix iF a
  = IIn (iF (IFix iF) a)
```

```
class IFunctor5 iF where
  imap5
    :: (∀a ... e.f a ... e → g a ... e)
    → iF f a ... e → iF g a ... e
newtype IFix5 iF a ... e
  = IIn5 (iF (IFix5 iF) a ... e)
```

Indexed Data types à la carte When building an eDSL one problem that becomes quickly prevalent is the so called *Expression Problem* [37]. A popular solution is *Data types à la carte* [35]: it combines constructors using the co-product of their signatures. This technique makes use of standard functors, however, an approach using IFunctors is described in *Compositional data types* [3]. This approach is upgraded further to add support for five type indices:

```
data (iF :+: iG) (f' :: i → j → k → l → m → ∗) (a :: i) ... (e :: m)
  = L :: iF f' a ... e → (iF :+: iG) f' a ... e | R :: iG f' a ... e → (iF :+: iG) f' a ... e
```

Using the :+: operator comes with problem of many L's and R's, when creating the AST. The solution, extended from [35] to also accommodate five type parameters, is to introduce a type class :≺: that injects them automatically.

Data types for each constructor can now be defined individually. The Then (⇐⇒) constructor is used as an example, however, the process can be applied to all constructors in the language.

```
data Then (iF :: [∗ → ∗] → [∗] →   [∗ → ∗] → [∗] → Nat → ∗)
          (insS  :: [∗ → ∗]) (insT  :: [∗])
          (outsS :: [∗ → ∗]) (outsT :: [∗]) (nins :: Nat) where
  Then :: (DataStore' fs as, DataStore' gs bs, DataStore' hs cs)
    ⇒ iF fs as gs bs nfs → iF gs bs hs cs ngs → Then iF fs as hs cs nfs
```

Each iF denotes the recursive points in the data type, with the subsequent type arguments mirroring those seen in Section 2.3. A corresponding $\mathsf{IFunctor}_5$ instance formalises the points of recursion, by describing how to transform the structure inside it. The smart constructor, that injects the L's and R's automatically can be defined for Then adding one extra constraint, to the constructor defined in Section 2.3 (Then :≺: iF), allowing the smart constructor to produce a node in the AST for any sum of data types, that includes the Then data type.

Representing a Circuit Once each constructor has been defined, they can be combined together to form the CircuitF type to represent a circuit. IFix_5 then ties the recursive knot to define the Circuit type.

```
type CircuitF = Id :+: Replicate :+: Then :+: ... :+: Task :+: Map
type Circuit = IFix5 CircuitF
```

Now that it is possible to build a Circuit, which can be considered a specification for how to execute a set of tasks, there needs to be a mechanism in place to execute the specification.

3.2 Network Typeclass

A Network represents a mechanism for executing the computation described by a Circuit. To allow for multiple execution mechanisms, a Network type class defines the key features each network requires:

```
class Network n where
  startNetwork :: Circuit insS insT outsS outsT nIns
               → IO (n   insS insT outsS outsT)
  stopNetwork  :: n insS insT outsS outsT → IO ()
  write        :: IHList insS insT        → n insS insT outsS outsT → IO ()
  read         :: n insS insT outsS outsT → IO (IHList outsS outsT)
```

This type class requires that a network has 4 different functions: startNetwork is responsible for converting the circuit into the underlying representation for a process network: it will be discussed in more detail in Section 3.4; stopNetwork is for cleaning up the network after it is no longer needed. For example, stopping any threads running. This could be particularly important if embedding a circuit into a larger program, where unused threads could be left hanging; write should take some input values and add them into the network, so that they can be processed; read should retrieve some output values from the network. nIns is required for the translation of Circuit to Network, therefore it is not inluded in the type of a network.

3.3 The Basic Network Representation

A BasicNetwork is an implementation of a Network that uses a Kahn Process Network (KPN). This means that each task in a circuit will run on its own separate thread, with inputs being passed between them on unbounded channels (from Control.Concurrent). A BasicNetwork stores the multiple input and output channels, to do so it leverages a special case of IHList.

```
data PipeList (fs :: [∗ → ∗]) (as :: [∗]) where
  PipeCons :: Chan (f a) → PipeList fs as → PipeList (f ': fs) (a ': as)
  PipeNil  :: PipeList '[] '[]
```

Using these PipeLists, BasicNetwork is defined using record syntax allowing for named fields, with accessors automatically generated.

```
data BasicNetwork (insS  :: [∗ → ∗]) (insT  :: [∗])
                  (outsS :: [∗ → ∗]) (outsT :: [∗]) where
  BasicNetwork :: {
    threads :: Map TaskUUID ThreadId,     -- allows threads to be managed
    jobs    :: Map JobUUID  JobStatus,    -- avoids duplicate job UUIDs
    ins     :: PipeList inpS  inpT,       -- to feed in inputs
    outs    :: PipeList outsS outsT       -- to retrieve outputs
  } → BasicNetwork inS insT outsS outsT
```

The Network type instance for a BasicNetwork is relatively trivial to implement using Control.Monad's **forM**_ if given a function to transform a Circuit to it.

```
instance Network BasicNetwork where
  startNetwork   = buildBasicNetwork   -- Defined soon...
  stopNetwork n = forM_ (threads n) killThread
  write uuid xs n = writePipes xs (ins n)
  read n          = readPipes (outs n)
```

The writePipes function will input a list of values into each of the respective pipes. The readPipes function will make a blocking call to each channel to read an output from it. This function will block till an output is read from every output channel.

3.4 Translation to a BasicNetwork

There is now a representation for a Circuit that the user will build, and a representation used to execute the Circuit. However, there is no mechanism to convert between them. This can be achieved by folding the circuit data type into a network. This fold, however, will need to create threads and channels, both of which are IO actions, and of course it will also need to deal with the numerous type parameters of Circuit. Such requirements lead to an exciting take on the *catamorphism* method for performing generalised folding of an abstract datatype.

Indexed Monadic Catamorphism The use of a catamorphism removes the recursion from any folding of the datatype. This means that the algebra can focus on one layer at a time. This also ensures that there is no re-computation of recursive calls, as this is all handled by the catamorphism. icata is able to fold an IFix iF a and produce an item of type f a. It uses the algebra argument as a specification of how to transform a single layer of the datatype. Normal catamorphisms can use monadic computations if defined as follows:

```
cataM :: (Traversable f, Monad m) ⇒ (∀a.f a → m a) → Fix f → m a
cataM algM (In x) = algM =≪ mapM (cataM algM) x
```

This monadic catamorphism [11] follows a similar pattern to a standard catamorphism, but instead uses functions such as a monadic map — mapM :: Monad m ⇒ (a → m b) → f a → m (f b). This allows the monadic catamorphism to be applied recursively on the data type being folded.

A similar technique can also be applied to indexed catamorphisms to gain a monadic version [3], however, to do so an indexed monadic map has to be introduced. imapM is the indexed equivalent of mapM, it performs a natural transformation, but is capable of also using monadic computation. This is included in the IFunctor type class, and facilitates the definition of icataM.

For Circuit, there is one final step that needs to be done: accommodating the five type parameters. To do this, IFunctor's imapM gets gifted the type parameters to complete the $\mathsf{IFunctor}_5$ class and allow the definition of icataM_5.

BuildNetworkAlg The final piece of the translation puzzle is an algebra for the fold. However, a standard algebra will not be able to complete this transformation. Consider an example Circuit with two tasks executed in sequence: task1 <=> task2. In a standard algebra, both sides of the Then constructor would be evaluated independently. In this case it would produce two disjoint networks, both with their own input and output channels. The algebra for Then, would then need to join the output channels of task1 with the input channels of task2. However, it is not possible to join channels together. Instead, the output channels from task1 need to be accessible when creating task2. This is referred to as a *context-sensitive* or *accumulating* fold. An accumulating fold forms series of nested functions, that collapse to give a final value once the base case has been applied. A simple example of an accumulating fold could be, implementing foldl in terms of foldr.

To be able to have an accumulating fold inside an indexed catamorphism a carrier data type is required to wrap up this function. This carrier, which shall be named AccuN, contains a function that when given a network that has been accumulated up to that point, then it is able to produce a network including the next layer in a circuit. This can be likened to the lambda function given to foldr, when defining foldl. The type of the layer being folded will be Circuit a b c d e.

```
newtype AccuN n asS asT a b c d e = AccuN
  {unAccuN :: n asS asT a b → IO (n asS asT c d)}
```

This newtype has two additional type parameters at the beginning, namely: asS and asT. They represent the input types to the initial circuit. Since the accumulating fold will work layer by layer from the top downwards, these types will remain constant and never change throughout the fold.

Classy Algebra To ensure that the approach remains modular, the algebra takes the form of a type class: the interpretation of a new constructor is just a new type class instance.

```
class (Network n, IFunctor5 iF) ⇒ BuildNetworkAlg n iF where
  buildNetworkAlg :: iF ( AccuN n asS asT) bsS bsT csS csT nbs
                  → IO ((AccuN n asS asT) bsS bsT csS csT nbs)
```

This algebra type class takes two parameters: n and iF. The n is constrained to have a Network instance, this allows the same algebra to be used for defining folds for multiple network types. The iF is the IFunctor that this instance is being defined for, an example is Then or Id. This algebra uses the AccuN data type to perform an accumulating fold. The input to the algebra is an IFunctor with the inner elements containing values of type AccuN. The function can be retrieved from inside AccuN to perform steps that are dependent on the previous, for example, in the Then constructor.

The Initial Network Given the use of an accumulating fold, one important question needs to be answered: what happens on the first layer? The fold needs an initialNetwork that has matching input and output types:

```
initialNetwork
  :: ∀insS insT.(InitialPipes insS insT) ⇒ IO (BasicNetwork insS insT insS insT)
initialNetwork = do
  ps ← initialPipes :: IO (PipeList insS insT)
  return (BasicNetwork empty empty ps ps)
```

The InitialPipes type class constructs an initialPipes based on the type required in the initial network.

The Translation Now that the algebra type class, and the initial input to the accumulating fold is defined, each instance of the type class can be defined.

Basic Constructors There are several constructors that just manipulate the output PipeList, these constructors are Id, Replicate, Swap, DropL, and DropR. The Swap constructor takes two inputs and then swaps them over:

```
instance BuildNetworkAlg BasicNetwork Swap where
  buildNetworkAlg Swap = return $ AccuN (λn → do
    let PipeCons c1 (PipeCons c2 PipeNil) = outs n
    return $ BasicNetwork
      (threads n) (jobs n) (ins n)
      (PipeCons c2 (PipeCons c1 PipeNil)))
```

The instance for Swap, defines a function wrapped by AccuN, that takes the current accumulated network, up to this point. It then transforms the outputs by swapping c1 and c2, and building a new BasicNetwork. All other leaf constructors will follow this pattern.

Task In a BasicNetwork, a task will run as a separate thread, to do this forkIO :: IO () → IO ThreadId will be used. Using this function requires some IO () computation to run, this will be defined by taskExecutor, which will read a value from each of input channels, execute the task with those inputs, and then write the output to the output channels. This computation is then repeated forever. Making use of the taskExecutor, the algebra instance for Task is as:

```
instance BuildNetworkAlg BasicNetwork Task where
  buildNetworkAlg (Task t) = return $ AccuN (λn → do
    out      ← PipeCons <$> newChan <*> return PipeNil
    taskUUID ← genUnusedTaskUUID (threads n)
    threadId ← forkIO (taskExecuter (Task t) taskUUID (outputs n) output)
    return $ BasicNetwork
      (M.insert taskUUID threadId (threads n)) (jobs n) (inputs n) output
```

This instance first creates a new output channel, this will be given to the task to send its outputs on. It then forks a new thread with the computation generated by taskExecutor. The executor is given the output values of the accumulated network and the output channel, just created. The resulting network has the same inputs, but now adds a new thread id to the list and the outputs set to be the output channels from the task.

Then The Then constructor is responsible for connecting circuits in sequence. When converting this to a network, this will involve making use of the accumulated network value to generate the next layer. The instance is defined as:

```
instance BuildNetworkAlg BasicNetwork Then where
  buildNetworkAlg (Then (AccuN fx) (AccuN fy))
    = return $ AccuN (fx >=> fy)
```

This instance has an interesting definition: firstly it takes the accumulated network n as input. It then uses the function fx, with the input n to generate a network for the top half of the Then constructor. Finally, it takes the returned network, from the top half of the constructor, and generates a network using the function fy representing the bottom half of the constructor.

Beside The Beside (<>) constructor places two circuits side by side. This is the most tedious algebra to define as the accumulated network needs to be split in half to pass to the two recursive sides of Beside. Details of its translation can be found in Appendix E [36].

CircuitFlow also uses the ExceptT monad transformer to fail gracefully.

4 Benchmarks

We use the audio streaming example from Section 2 to perform the benchmarking. It is also the main application domain of Luigi which we will compare with.

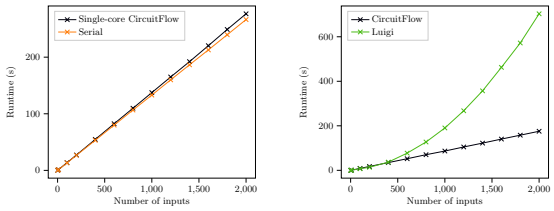
Haskell benchmarks were taken using criterion [28]; Python 3.8.5 benchmarks with pytest − benchmark [22]. Each benchmark is tested on thirteen different numbers of inputs: 1, 10, 100, 200, then at intervals of 200 until 2000, with measurements repeated and summarised as a mean average.

Three months of one of the author's own audio history is used, to ensure that the data closely aligns with the real world. This allows for the evaluation of how each implementation scales with more inputs. All benchmarks take place on an Intel(R) Core(TM) i5-4690 CPU at 3.50GHz (4 cores and no hyper-threading), with 8GB of RAM booting Ubuntu 20.04.

Multi-Core Haskell By default the Haskell runtime does not enable multi-core processing. Considering the aim of this project partly involves making CircuitFlow run in parallel, multi-core processing is crucial. To enable this the `-threaded` flag is set when building the binary. Then, using the runtime options, the number of threads can be set by adding `+RTS -N` flags when running the binary. The `-N` allows the runtime to select the optimal number of threads for the program.

Parallel vs Serial The first test will ensure that CircuitFlow's parallelisation has a positive effect on run-times. To ensure that the test is fair, the serial implementation will make use of the same tasks in the pre-processing pipeline. The inputs and outputs will just be manually fed into each task, in a sequential way. The results show that CircuitFlow does indeed provide a performance gain, with a mean speedup of 1.53x.

Profiling the circuit shows that a significant proportion of time is spend reading CSV files. Optimising speed of CSV parsing and how often a CSV is read via caching would improve runtime. Another area for improvement is that there is an expectation on the user to know where is best to split up the workflow into tasks. It would be beneficial if a circuit could automatically fuse tasks together, then it would have a positive effect on the runtime.



(a) Linear (b) Vs Luigi

Fig. 3: CircuitFlow benchmarks

1 Core Circuit vs Serial Another interesting scenario to test is checking if the network structure adds additional overhead, in a situation where there is only 1 core. To test this, the multi-core support of the Haskell runtime will not be enabled: this will then simulate multiple cores with context switching. Figure 3a, shows the results of this benchmark. It shows that both the linear and single core implementation scale together in a linear fashion. Most importantly, CircuitFlow only adds a minor overhead over a linear implementation. This will be particularly helpful for a user which needs to run code on multiple types of devices. There is no need for them to create a different implementation for devices where parallelisation may not be possible.

CircuitFlow vs Luigi The final benchmark on CircuitFlow is comparing it to widely used library: Luigi by Spotify [32]. Since Luigi uses a Data Process Network (DPN), it can use any number of threads: in this test it is set to 4 — the same as CircuitFlow. Figure 3b, shows the results of the benchmark.

This shows that CircuitFlow performs better than Luigi on larger numbers of inputs. CircuitFlow scales linearly with the number of inputs, whereas Luigi's runtime appears to grow at a quicker rate than linear.

Why is CircuitFlow so good? Luigi and CircuitFlow have their differences, which will likely explain why there is a difference in run times, especially with larger numbers of inputs.

More Lightweight Luigi is a far more complex library with advanced features, not included in CircuitFlow, that may slow Luigi down — one such feature is back filling. This allows Luigi to avoid running tasks that have already been run. This feature means that before executing a task the Luigi scheduler has to check if a task has already been executed. This adds additional overhead to the scheduler that CircuitFlow does not have. Although this feature does have its benefits, after the first run of Luigi all run times after are very quick as no tasks will need to be executed. If CircuitFlow were to implement this feature any overhead it adds will be partially mitigated by the checks being distributed across multiple threads, instead of in one central scheduler.

Computation Models The two libraries use variants of the same computation model: CircuitFlow uses a KPN and Luigi uses a DPN [21]. This difference is the main reason why CircuitFlow scales linearly when it needs to process more input values. CircuitFlow makes use of buffered channels to keep a queue of all inputs that need to be processed. However, Luigi does not rely on this design, instead it has a pool of workers with a scheduler controlling what is executed on each worker. It is this scheduler that causes Luigi to scale non-linearly. As the number of inputs grow, the scheduler will have to schedule more and more tasks: this process is not $\mathcal{O}(n)$.

Multi-processing in Python CircuitFlow makes use of a static number of threads defined by the number of tasks in a circuit. Luigi on the other hand can support any number of workers, however, Luigi suffers from a downfall of Python: threads cannot run in parallel due to the Global Interpreter Lock. To avoid this Luigi uses processes not threads, which adds extra overhead. Luigi also creates a new process for each invocation of a task, which CircuitFlow does not do. This means that Luigi will start 8000 processes vs CircuitFlow's 4 threads for the 2000 inputs benchmark. CircuitFlow's static number of threads could also be considered a downside due to the lack of flexibility depending on run-time values. To combat this more combinators can be introduced that allow for branching or other similar operations, in fact, mapC is a combinator of this type.

5 Discussion and Related Work

In this section, we cover the embedding techniques that we build upon and how our process can be replicated. We also discuss other popular workflow libraries including imperative and functional ones, comparing them to CircuitFlow.

Summary of Embedding Techniques and their General Use. CircuitFlow, which can be more generally be seen as an eDSL whose semantics needs to use effects and has rich types to verify program correctness, has been created in a modular manner that doesn't compromise on performance. The pivotal parts of CircuitFlow's creation can be replicated to produce such other eDSLs. The process is one of three parts. The first is the curation of the type information. In the case of CircuitFlow, this was dependency information, and was achieved using Haskell's approximation of dependant types (DataKinds [39] and Singletons [9] for value promotion/demotion to/from the type level; Type Families [31] for type information manipulation; and Heterogeneous lists [18] for, well, storing more than one type in a list). The second act follows the same beats as the classic embedding story [12]: each construct is created as a separate fixed functor, where all constructs can be composed together with the beloved *Data types à la carte*, and semantics provided through a "classy" algebra. The story just needs to be jacked up to accommodate the type information and effects, with the trick being the switch to indexed functors [25, 17] and a monadic catamorphism [11]. Finally, the choice of underlying semantics is key for speed as that is ultimately what will be running. Our choice of KPNs assisted us greatly with CircuitFlow's competitive run-time.

Applicative Functors An example of capturing parallelism in Haskell is to use applicative functors [26] — a technique employed by the Haxl library [23]. This approach can leverage the applicative combinators to group together computation that can be performed simultaneously. There is even the ApplicativeDo language extension [24], which desugars do notation down to applicative combinators. However, this approach suffers from some forced sequentiality at points. Take the previously mentioned example Figure 1, both top ten tasks would be

grouped together. Leading to neither task being able to begin until both aggregations have been completed.

Arrows Another method is arrows [15], used by Funflow based on *Composing Effects into Tasks and Workflows* [29]. Arrows similarly are often used through with the notation obtained from the language extension [30], which introduces a do style notation. They also fall victim to the same problems as applicative functors. Due to the constructor arr consuming a function, it is not possible to inspect inside and fully exploit all cases of parallelisation.

The Funflow library that makes use of arrows, does so by noticing that tasks in workflows are similar to effects in the functional community. It draws from existing work on combining and analysing effects, with categories and arrows, and applies this to constructing workflows.

Symmetric Monoidal Categories (SMCs) Linear Haskell is put to excellent use in *Evaluating Linear Functions to Symmetric Monoidal Categories* [4] to address the problem of over sequentialisation found in applicative functors and arrows. It introduces a new SMC type class that allows for all parallelism to be exposed and exploited in a workflow. The type class adds new combinators for linear Haskell functions, that can be composed in a style that aligns with do notation. It uses atomic types to detail the synchronisation points, and where synchronisation can be discovered by a scheduler. However, it comes with the caveat that it can only compose linear functions.

Pipes [13] focuses on supporting steaming data, which is beneficial as there is no need to wait for jobs to finish before moving on. This is something that CircuitFlow is also designed to support without any modifications: a network can be started and inputs can be streamed in when they are available.

Luigi [32] Industry-favourite Luigi, used to orchestrate tasks in a data workflow, is a library that, as we have seen, falls into the trap of un-typed task dependencies. It makes use of a central scheduler and workers, allowing work to be distributed across multiple machines. It also comes with built-in support for many different output formats, such as files in a Hadoop file system.

SciPipe [20] An approach for orchestrating external jobs is taken by SciPipe, a workflow library for agile development of complex and dynamic bioinformatics pipelines. Unlike CircuitFlow and many other libraries, instead of defining tasks as functions within the embedding language, SciPipe uses Bash commands to easily interact with pre-existing binaries. This allows task to be written in the language most suited for its requirements, however, comes with the downside of the additional infrastructure required to create all these binaries for each task. Due to the separation of tasks into bash scripts, type checking interactions between tasks is significantly harder.

Other Typed Dataflow Libraries DryadLINQ [40] allows for developers to create parallel programs in SQL-like LINQ expressions. Similarly to CircuitFlow, these can be inspected to find any data-parallel sections and then automatically translated into a distributed execution plan that can run on Dryad — although CircuitFlow currently lacks a distributed network implementation. FlumeJava [5], uses lazy evaluation of operations on parallel data structures, to build a dataflow graph of the steps required. When the value is required the graph is optimised to evaluate the operations in an optimal way. Unlike CircuitFlow, Naiad [27] can execute cyclic dataflow programs. It does so on a distributed system, to help with streaming data analysis or iterative machine learning training.

Staged Selective Parser Combinators [38] Indexed functors [25], are a new technique for building typed eDSL. This paper makes use of this new tool to have a type index representing the type of a parser. This allows it to make optimisations and translations while ensuring that the value parsed never changes.

6 Conclusion

This paper introduced a new eDSL to declaratively construct data workflows, which are type-safe and competitive in run-time performance. The design of CircuitFlow draws its origins from a strong mathematical background, with each constructor directly representing an axiom in a SMP. This demonstrates the language's completeness at being able to represent any DAG, that a data workflow may need. The battle for type-safety without compromising run-time or modular design was a tough one, but one that can be replicated to great avail when creating languages with a similar requirements.

Acknowledgements. The authors would like to thank Jamie Willis for his insights while creating CircuitFlow and the anonymous reviewers for their constructive and helpful comments.

The work is partly supported by EPSRC Grant *EXHIBIT: Expressive High-Level Languages for Bidirectional Transformations* (EP/T008911/1) and Royal Society Grant *Bidirectional Compiler for Software Evolution* (IES370104).

References

1. Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., Kudlur, M., Levenberg, J., Monga, R., Moore, S., Murray, D.G., Steiner, B., Tucker, P., Vasudevan, V., Warden, P., Wicke, M., Yu, Y., Zheng, X.: Tensorflow: A system for large-scale machine learning. In: 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16). pp. 265–283. USENIX Association, Savannah, GA (Nov 2016)
2. Apache: Airflow, http://airflow.apache.org

3. Bahr, P., Hvitved, T.: Compositional data types. In: Proceedings of the Seventh ACM SIGPLAN Workshop on Generic Programming. p. 83–94. WGP '11, Association for Computing Machinery, New York, NY, USA (2011)
4. Bernardy, J.P., Spiwack, A.: Evaluating linear functions to symmetric monoidal categories. In: Proceedings of the 14th ACM SIGPLAN International Symposium on Haskell. p. 14–26. Haskell 2021, Association for Computing Machinery, New York, NY, USA (2021)
5. Chambers, C., Raniwala, A., Perry, F., Adams, S., Henry, R., Bradshaw, R., Nathan: Flumejava: Easy, efficient data-parallel pipelines. In: ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI). pp. 363–375. 2 Penn Plaza, Suite 701 New York, NY 10121-0701 (2010)
6. Coecke, B., Fritz, T., Spekkens, R.W.: A mathematical theory of resources. Information and Computation **250**, 59–86 (Oct 2016)
7. Dennis, J.B., Misunas, D.P.: A preliminary architecture for a basic data-flow processor. In: Proceedings of the 2nd Annual Symposium on Computer Architecture. p. 126–132. ISCA '75, Association for Computing Machinery, New York, NY, USA (1974)
8. Eisenberg, R.A., Vytiniotis, D., Peyton Jones, S., Weirich, S.: Closed type families with overlapping equations. In: Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. p. 671–683. POPL '14, Association for Computing Machinery, New York, NY, USA (2014)
9. Eisenberg, R.A., Weirich, S.: Dependently typed programming with singletons. In: Proceedings of the 2012 Haskell Symposium. p. 117–130. Haskell '12, Association for Computing Machinery, New York, NY, USA (2012)
10. Erdmann, M., Fischer, B., Fischer, R., Rieger, M.: Design and execution of make-like, distributed analyses based on spotify's pipelining package luigi. Journal of Physics: Conference Series **898**, 072047 (Oct 2017)
11. Fokkinga, M.: Monadic maps and folds for arbitrary datatypes. Memoranda informatica (94-28), – (Jun 1994), imported from EWI/DB PMS [db-utwente:tech:0000003538]
12. Gibbons, J., Wu, N.: Folding domain-specific languages: Deep and shallow embeddings (functional pearl). Proceedings of the ACM SIGPLAN International Conference on Functional Programming, ICFP **49** (Aug 2014)
13. Gonzalez, G.: pipes, https://hackage.haskell.org/package/pipes
14. Hils, D.D.: Visual languages and computing survey: Data flow visual programming languages. J. Vis. Lang. Comput. **3**, 69–101 (1992)
15. Hughes, J.: Generalising monads to arrows. Science of Computer Programming **37**(1), 67–111 (2000)
16. Inc, A.: Quartz composer user guide (Jul 2007)
17. Johann, P., Ghani, N.: Foundations for structured programming with gadts. In: Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. p. 297–308. POPL '08, Association for Computing Machinery, New York, NY, USA (2008)
18. Kiselyov, O., Lämmel, R., Schupke, K.: Strongly typed heterogeneous collections. In: Proceedings of the 2004 ACM SIGPLAN Workshop on Haskell. p. 96–107. Haskell '04, Association for Computing Machinery, New York, NY, USA (2004)
19. Kotliar, M., Kartashov, A.V., Barski, A.: CWL-Airflow: a lightweight pipeline manager supporting Common Workflow Language. GigaScience **8**(7) (Jul 2019), giz084

20. Lampa, S., Dahlö, M., Alvarsson, J., Spjuth, O.: SciPipe: A workflow library for agile development of complex and dynamic bioinformatics pipelines. GigaScience **8**(5) (Apr 2019), giz044
21. Lee, E.A., Parks, T.M.: Dataflow process networks. Proceedings of the IEEE **83**(5), 773–801 (1995)
22. Maries, I.C.: time, https://pypi.org/project/pytest-benchmark/
23. Marlow, S., Brandy, L., Coens, J., Purdy, J.: There is no fork: An abstraction for efficient, concurrent, and concise data access. In: Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming. p. 325–337. ICFP '14, Association for Computing Machinery, New York, NY, USA (2014)
24. Marlow, S., Peyton Jones, S., Kmett, E., Mokhov, A.: Desugaring haskell's donotation into applicative operations. SIGPLAN Not. **51**(12), 92–104 (Sep 2016)
25. McBride, C.: Functional pearl: Kleisli arrows of outrageous fortune. Journal of Functional Programming (accepted for publication) (2011)
26. Mcbride, C., Paterson, R.: Applicative programming with effects. J. Funct. Program. **18**(1), 1–13 (Jan 2008)
27. Murray, D., McSherry, F., Isaacs, R., Isard, M., Barham, P., Abadi, M.: Naiad: A timely dataflow system. In: Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP). p. 439–455. ACM (Nov 2013)
28. O'Sullivan, B.: criterion, http://www.serpentine.com/criterion/
29. Parès, Y., Bernardy, J.P., Eisenberg, R.A.: Composing effects into tasks and workflows. In: Proceedings of the 13th ACM SIGPLAN International Symposium on Haskell. p. 80–94. Haskell 2020, Association for Computing Machinery, New York, NY, USA (2020)
30. Paterson, R.: A new notation for arrows. In: International Conference on Functional Programming. pp. 229–240. ACM Press (Sep 2001)
31. Schrijvers, T., Peyton Jones, S., Chakravarty, M., Sulzmann, M.: Type checking with open type functions. In: Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming. p. 51–62. ICFP '08, Association for Computing Machinery, New York, NY, USA (2008)
32. Spotify: Spotify: Luigi, https://github.com/spotify/luigi
33. Spotify: Tasks (Apr 2020), https://luigi.readthedocs.io/en/stable/tasks.html
34. Svenningsson, J., Axelsson, E.: Combining deep and shallow embedding of domain-specific languages. Computer Languages, Systems & Structures **44**, 143–165 (2015), sI: TFP 2011/12
35. Swierstra, W.: Data types à la carte. Journal of Functional Programming **18**(4), 423–436 (2008)
36. TODO: Circuitflow: A domain specific language for dataflow programming (with appendices)
37. Wadler, P.: The expression problem (Nov 1998)
38. Willis, J., Wu, N., Pickering, M.: Staged selective parser combinators. Proc. ACM Program. Lang. **4**(ICFP), 1–30 (Aug 2020)
39. Yorgey, B.A., Weirich, S., Cretin, J., Peyton Jones, S., Vytiniotis, D., Magalhães, J.P.: Giving haskell a promotion. In: Proceedings of the 8th ACM SIGPLAN Workshop on Types in Language Design and Implementation. p. 53–66. TLDI '12, Association for Computing Machinery, New York, NY, USA (2012)
40. Yu, Y., Isard, M., Fetterly, D., Budiu, M., Erlingsson, U., Gunda, P.K., Currey, J.: Dryadlinq: A system for general-purpose distributed data-parallel computing using a high-level language. In: Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation. p. 1–14. OSDI'08, USENIX Association, USA (2008)

41. Zaharia, M., Chowdhury, M., Franklin, M.J., Shenker, S., Stoica, I.: Spark: Cluster computing with working sets. In: Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing. p. 10. HotCloud'10, USENIX Association, USA (2010)