Dissertation Type: research

DEPARTMENT OF COMPUTER SCIENCE

CircuitFlow: A Domain Specific Language for Dataflow Programming

Riley Evans

A dissertation submitted to the University of Bristol in accordance with the requirements of the degree of Master of Engineering in the Faculty of Engineering.

Sunday 9th May, 2021

Declaration

This dissertation is submitted to the University of Bristol in accordance with the requirements of the degree of MEng in the Faculty of Engineering. It has not been submitted for any other degree or diploma of any examining body. Except where specifically acknowledged, it is all the work of the Author.

Riley Evans, Sunday 9th May, 2021

Contents

List of Figures

Notation and Acronyms

DSL Domain Specific Language

EDSL Embedded DSL

FIFO First-In First-Out

KPN Kahn Process Network

GPL General Purpose Language

DPN Data Process Network

DAG Directed Acyclic Graph

AST Abstract Syntax Tree

PID Process Identifier

Acknowledgements

Change this to something meaningful

It is common practice (although totally optional) to acknowledge any third-party advice, contribution or influence you have found useful during your work. Examples include support from friends or family, the input of your Supervisor and/or Advisor, external organisations or persons who have supplied resources of some kind (e.g., funding, advice or time), and so on.

Chapter 1

Introduction

Write an introduction (do near the end)

Chapter 2

Background

2.1 Dataflow Programming

Dataflow programming is a paradigm that models applications as a directed graph. The nodes of the graph have inputs and outputs and are pure functions, therefore have no side effects. It is possible for a node to be a: source; sink; or processing node. A source is a read-only storage: it can be used to feed inputs into processes. A sink is a write-only storage: it can be used to store the outputs of processes. Processes will read from either a source or the output of another process, and then produce a result which is either passed to another process or saved in a sink. Edges connect these nodes together, and define the flow of information.

Example - Data Pipelines A common use of dataflow programming is in pipelines that process data. This paradigm is particularly helpful as it helps the developer to focus on each specific transformation on the data as a single component. Avoiding the need for long and laborious scripts that could be hard to maintain. One example of a data pipeline tool that makes use of dataflow programming is Luigi [22]. An example dataflow graph produced by the tool is shown in Figure 2.1

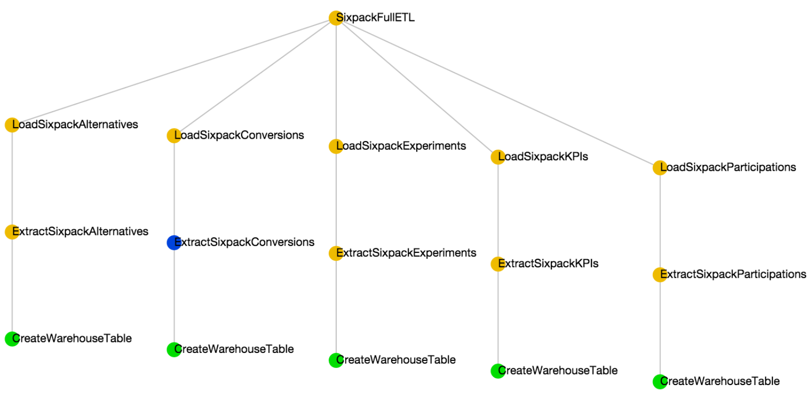


Figure 2.1: Luigi dependency graph [11]

Example - Quartz Composer Apple developed a tool included in XCode, named Quartz Composer, which is a node-based visual programming language [1]. As seen in Figure 2.2, it uses a visual approach to programming connecting nodes with edges. This allows for quick development of programs that process and render graphical data, without the user having to write a single line of code. This means that even non-programmers are able to use the tool.

Example - Spreadsheets A widely used example of dataflow programming is in spreadsheets. A cell in a spreadsheet can be thought of as a single node. It is possible to specify dependencies to other cells through the use of formulas. Whenever a cell is updated it sends its new value to those who depend on it, and so on. Work has also done to visualise spreadsheets using dataflow diagrams, to help debug ones that are complex [9].

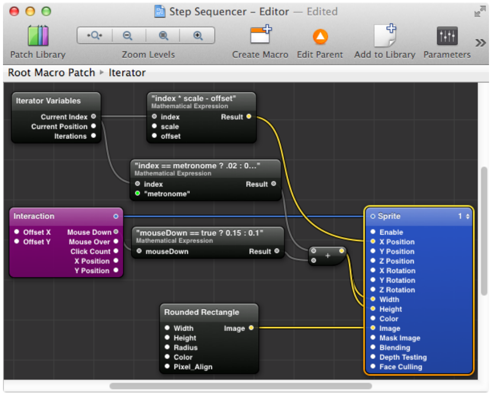


Figure 2.2: Quartz composer [6]

2.1.1 The Benefits

Visual The dataflow paradigm uses graphs, which make programming visual. It allows the end-user programmer to see how data passes through the program, much easier than in an imperative approach. In many cases, dataflow programming languages use drag and drop blocks with a graphical user interface to build programs. For example, Tableau Prep [26], that makes programming more accessible to users who do not have programming skills.

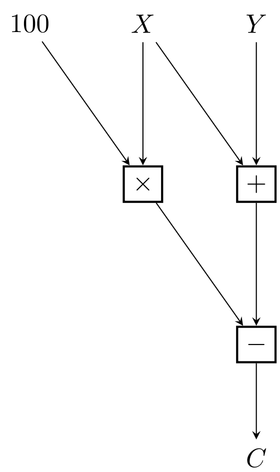
Implicit Parallelism Moore's law states that the number of transistors on a computer chip doubles every two years [19]. This meant that the chips' processing speeds also increased in alignment with Moore's law. However, in recent years this is becoming harder for chip manufacturers to achieve [3]. Therefore, chip manufactures have had to turn to other approaches to increase the speed of new chips, such as multiple cores. It is this approach the dataflow programming can effectively make use of. Since each node in a dataflow is a pure function, it is possible to parallelise implicitly. No node can interact with another node, therefore there are no data dependencies outside of those encoded in the dataflow. Thus eliminating the ability for a deadlock to occur.

2.1.2 Dataflow Diagrams

Dataflow programs are typically viewed as a graph. An example dataflow graph along with its corresponding imperative approach, can be found in Figure 2.3. The nodes 100, X, and Y are sources as they are only read from. C is a sink as it is wrote to. The remaining nodes are all processes, as they have some number of inputs and compute a result.

$$
\begin{aligned}
A &:= 100 \times X \\
B &:= X + Y \\
C &:= A - B
\end{aligned}
$$

(a)



(b)

Figure 2.3: An example dataflow and its imperative approach.

In this diagram is possible to see how implicit parallelisation is possible. Both A and B can be calculated simultaneously, with C able to be evaluated after they are complete.

2.1.3 Kahn Process Networks (KPNs)

A method introduced by Gilles Kahn, Kahn Process Networks (KPNs) realised the concept of dataflow networks through the use of threads and unbounded First-In First-Out (FIFO) queues [13]. The FIFO queue is one where the items are output in the same order that they are added. A node in the dataflow becomes a thread in the process network. Each FIFO queue represents the edges connecting the nodes in a graph. The threads are then able to communicate through FIFO queues. The node can have multiple input queues and is able to read any number of values from them. It will then compute a result and add it to an output queue. Kahn imposed a restriction on a process in a KPNs that the thread is suspended if it attempts to fetch a value from an empty queue. The thread is not allowed to test for the presence of data in a queue.

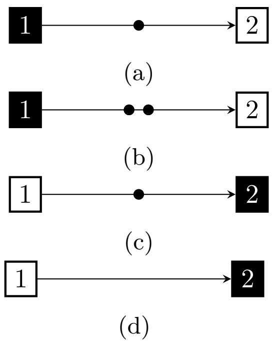


Figure 2.4: A sequence of node firings in a KPN

Parks described a variant of KPNs, called Data Process Networks (DPNs) [16]. They recognise that if functions have no side effects then they have no values to be shared between each firing. Therefore, a pool of threads can be used with a central scheduler instead.

2.2 Domain Specific Languages (DSLs)

A DSL is a programming language that has a specialised domain or use-case. This differs from a General Purpose Language (GPL), which can be applied across a larger set of domains, and are generally turing complete. HTML is an example of a DSL: it is good for describing the appearance of websites, however, it cannot be used for more generic purposes, such as adding two numbers together.

Approaches to Implementation DSLs are typically split into two categories: standalone and embedded. Standalone DSLs require their own compiler and typically their own syntax; HTML would be an example of a standalone DSL. Embedded DSLs (EDSLs) use an existing language as a host, therefore they use the syntax and compiler from the host. This means that they are easier to maintain and often quicker to develop than standalone DSLs. An EDSL, can be implemented using two differing techniques: deep and shallow embeddings.

2.2.1 Deep Embeddings

A deep embedding is when the terms of the DSL will construct an Abstract Syntax Tree (AST) as a host language datatype. Semantics can then be provided later on with evaluation functions. Consider the example of a minimal non-deterministic parser combinator library [29], which will be a running example for this chapter.

$$\begin{array}{l} \mathbf{data}\ \mathsf{Parser_d}\ (\mathsf{a} :: \mathsf{Type})\ \mathbf{where} \\ \quad \mathsf{Satisfy_d} :: (\mathsf{Char} \rightarrow \mathsf{Bool}) \rightarrow \mathsf{Parser_d}\ \mathsf{Char} \\ \quad \mathsf{Or_d} \quad\;\; :: \mathsf{Parser_d}\ \mathsf{a} \rightarrow \mathsf{Parser_d}\ \mathsf{a} \rightarrow \mathsf{Parser_d}\ \mathsf{a} \end{array}$$

This can be used to build a parser that can parse the characters `'a'` or `'b'`.

$\mathsf{aorb_d} :: \mathsf{Parser_d\ Char}$
$\mathsf{aorb_d} = \mathsf{Satisfy_d}\ (\equiv \texttt{'a'})\ \grave{}\,\mathsf{Or_d}\,\grave{}\ \mathsf{Satisfy_d}\ (\equiv \texttt{'b'})$

However, this parser does not have any semantics, therefore this needs to be provided by the evaluation function parse.

$\mathsf{parse_d} :: \mathsf{Parser_d\ a} \to \mathsf{String} \to [(\mathsf{a}, \mathsf{String})]$
$\mathsf{parse_d}\ (\mathsf{Satisfy_d\ p}) = \lambda\mathbf{case}$
$\quad [\,] \quad\quad \to [\,]$
$\quad (\mathsf{t} : \mathsf{ts'}) \to [(\mathsf{t}, \mathsf{ts'}) \mid \mathsf{p\ t}]$
$\mathsf{parse_d}\ (\mathsf{Or_d\ px\ py}) = \lambda\mathsf{ts} \to \mathsf{parse_d\ px\ ts} +\!\!+ \mathsf{parse_d\ py\ ts}$

The program can then be evaluated by the $\mathsf{parse_d}$ function. For example, $\mathsf{parse_d\ aorb_d}$ `"a"` evaluates to $[(\texttt{'a'}, \texttt{""})]$, and $\mathsf{parse_d\ aorb_d}$ `"c"` evaluates to $[\,]$.

A key benefit for deep embeddings is that the structure can be inspected, and then modified to optimise the user code: Parsley [28] makes use of such techniques to create optimised parsers. Another benefit, is that you can provide multiple interpretations, by specifying different evaluation functions. However, they also have drawbacks - it can be laborious to add a new constructor to the language. Since it requires that all functions that use the deep embedding be modified to add a case for the new constructor [24].

2.2.2 Shallow Embeddings

In contrast, a shallow approach is when the terms of the DSL are defined as first class components of the language. For example, a function in Haskell. Components can then be composed together and evaluated to provide the semantics of the language. Again a simple parser example can be considered.

$\mathbf{newtype}\ \mathsf{Parser_s\ a} = \mathsf{Parser_s}\ \{\mathsf{parse_s} :: \mathsf{String} \to [(\mathsf{a}, \mathsf{String})]\}$
$\mathsf{or_s} :: \mathsf{Parser_s\ a} \to \mathsf{Parser_s\ a} \to \mathsf{Parser_s\ a}$
$\mathsf{or_s}\ (\mathsf{Parser_s\ px})\ (\mathsf{Parser_s\ py}) = \mathsf{Parser_s}\ (\lambda\mathsf{ts} \to \mathsf{px\ ts} +\!\!+ \mathsf{py\ ts})$
$\mathsf{satisfy_s} :: (\mathsf{Char} \to \mathsf{Bool}) \to \mathsf{Parser_s\ Char}$
$\mathsf{satisfy_s\ p} = \mathsf{Parser_s}\ (\lambda\mathbf{case}$
$\quad [\,] \quad\quad \to [\,]$
$\quad (\mathsf{t} : \mathsf{ts'}) \to [(\mathsf{t}, \mathsf{ts'}) \mid \mathsf{p\ t}])$

The same $\mathsf{aorb_s}$ parser can be constructed from these functions, avoiding the need for an intermediate AST.

$\mathsf{aorb_s} :: \mathsf{Parser_s\ Char}$
$\mathsf{aorb_s} = \mathsf{satisfy_s}\ (\equiv \texttt{'a'})\ \grave{}\,\mathsf{or_s}\,\grave{}\ \mathsf{satisfy_s}\ (\equiv \texttt{'b'})$

Using a shallow implementation has the benefit of being able add new 'constructors' to a DSL, without having to modify any other functions. Since each 'constructor', produces the desired result directly. However, this causes one of the main disadvantages of a shallow embedding - the structure cannot be inspected. This means that optimisations cannot be made to the structure before evaluating it.

2.3 Higher Order Functors

It is possible to capture the shape of an abstract datatype as a Functor. The use of a Functor allows for the specification of where a datatype recurses. Consider an example on a small expression language:

$\mathbf{data}\ \mathsf{Expr} = \mathsf{Add\ Expr\ Expr}$
$\quad\quad\quad\quad \mid \mathsf{Val\ \ Int}$

The recursion within the Expr datatype can be removed to form ExprF. The recursive steps can then be specified in the Functor instance.

$\mathbf{data}\ \mathsf{ExprF\ f} = \mathsf{AddF\ f\ f}$
$\quad\quad\quad\quad\quad \mid \mathsf{ValF\ \ Int}$

```
instance Functor ExprF where
  fmap f (AddF x y) = AddF (f x) (f y)
  fmap f (ValF x)   = ValF  x
```

To regain a datatype that is isomorphic to the original datatype, the recursive knot need to be tied. This can be done with Fix, to get the fixed point of ExprF:

```
data Fix f  = In (f (Fix f))
type Expr' = Fix ExprF
```

There is, however, one problem: a Functor expressing the a parser language is required to be typed. Parsers require the type of the tokens being parsed. For example, a parser reading tokens that make up an expression could have the type Parser Expr. A Functor does not retain this type information needed in a parser.

IFunctors Instead a type class called IFunctor [17] — also known as HFunctor [12] — can be used, which is able to maintain the type indicies. This makes use of $\rightsquigarrow$, which represents a natural transformation [15] from f to g. 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. Whereas a functor changes the values inside a structure.

```
type (~>) f g = ∀a.f a → g a
class IFunctor iF where
  imap :: (f ~> g) → iF f ~> iF g
```

The shape of Parser can be seen in ParserF where the f marks the recursive spots. The type f represents the type of the children of that node. In most cases this will be itself.

```
data ParserF (f :: * → *) (a :: *) where
  SatisfyF :: (Char → Bool) → ParserF f Char
  OrF      :: f a → f a → ParserF f a
```

An IFunctor instance can be defined, which follow the same structure as a standard Functor instance.

```
instance IFunctor ParserF where
  imap _ (SatisfyF s) = SatisfyF s
  imap f  (OrF px py) = OrF (f px) (f py)
```

Fix is used to get the fixed point of a Functor, to get the indexed fixed point IFix can be used.

```
newtype IFix iF a = IIn (iF (IFix iF) a)
```

The fixed point of ParserF is $\text{Parser}_{\text{fixed}}$.

```
type Parser_fixed = IFix ParserF
```

In a deep embedding, the AST can be traversed and modified to make optimisations, however, it may not be the best representation when evaluating it. This means that it might be transformed to a different representation. In the case of a parser, this could be a stack machine. Now that the recursion in the datatype has been generalised, it is possible to create a mechanism to perform this transformation. An indexed *catamorphism* is one such way to do this, it is a generalised way of folding an abstract datatype. 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. The commutative diagram below describes how to define a catamorphism, that folds an IFix iF a to a f a.

$$\begin{tikzcd}
\text{iF (IFix iF) a} \arrow[r, "\text{imap (icata alg)}"] \arrow[d, shift left, "\text{IIn}"] & \text{iF f a} \arrow[d, "\text{alg}"] \\
\text{IFix iF a} \arrow[u, shift left, "\text{inop}"] \arrow[r, "\text{icata alg}"] & \text{f a}
\end{tikzcd}$$

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.

$$\mathsf{icata} :: \mathsf{IFunctor\ iF} \Rightarrow (\mathsf{iF\ f} \rightsquigarrow \mathsf{f}) \rightarrow \mathsf{IFix\ iF} \rightsquigarrow \mathsf{f}$$
$$\mathsf{icata\ alg\ (IIn\ x)} = \mathsf{alg\ (imap\ (icata\ alg)\ x)}$$

The resulting type of icata is f a, therefore the f has kind $* \rightarrow *$. This could be IFix ParserF, which would be a transformation to the same structure, possibly applying optimisations to the AST.

2.3.1 Monadic Catamorphism with IFunctors

Using an indexed catamorphism, allows for principled recursion and makes it easier to define a fold over a data type, as any recursive step is abstracted from the user. However, there may be times when the there is a need for monadic computations in the algebra. To be able to do this a monadic catamorphism is defined:

$$\mathsf{cataM} :: (\mathsf{Traversable\ f}, \mathsf{Monad\ m}) \Rightarrow (\forall \mathsf{a.f\ a} \rightarrow \mathsf{m\ a}) \rightarrow \mathsf{Fix\ f} \rightarrow \mathsf{m\ a}$$
$$\mathsf{cataM\ algM\ (In\ x)} = \mathsf{algM} \ll\!\!= \mathsf{mapM\ (cataM\ algM)\ x}$$

This catamorphism follows a similar pattern to a standard catamorphism, however, it does not have the Functor constraint. Instead it constrains f by Traversable: this type class still requires that f is a functor, it just provides additional functions such as a monadic map — $\mathsf{mapM} :: \mathsf{Monad\ m} \Rightarrow (\mathsf{a} \rightarrow \mathsf{m\ b}) \rightarrow \mathsf{f\ a} \rightarrow \mathsf{m\ (f\ b)}$. This allows the monadic catamorphism to be applied recursively on the data type being folded.

This technique can also be applied to indexed catamorphisms, however to do so an indexed monadic map has to be introduced. This will be included as part of the IFunctor type class.

```
class IFunctor iF where
  imap  :: (f ⤳ g) → iF f ⤳ iF g
  imapM :: Monad m ⇒ (∀a.f a → m (g a)) → iF f a → m (iF g a)
```

imapM is the indexed equivalent of mapM, it performs a natural transformation, but is capable of also using monadic computation.

The new IFunctor instance for ParserF is defined as:

```
instance IFunctor ParserF where
  imap = ...   -- already defined
  imapM _ (SatisfyF s) = return SatisfyF s
  imapM f (OrF px py) = do
    px' ← f px
    py' ← f py
    return (OrF px' py')
```

The definition for imapM on ParserF is intuitively the same, however just uses do-notation instead. Making use of imapM, icataM is defined to be:

$$\mathsf{icataM} :: (\mathsf{IFunctor\ iF}, \mathsf{Monad\ m}) \Rightarrow (\forall \mathsf{a.iF\ f\ a} \rightarrow \mathsf{m\ (f\ a)}) \rightarrow \mathsf{IFix\ iF\ a} \rightarrow \mathsf{m\ (f\ a)}$$
$$\mathsf{icataM\ algM\ (IIn\ x)} = \mathsf{algM} \ll\!\!= \mathsf{imapM\ (icataM\ algM)\ x}$$

icataM, has a similar structure to all other catamorphisms defined, however it takes a monadic algebra, that can be used to transform the structure of the input type.

2.4 Data types à la carte

When building a DSL one problem that becomes quickly prevalent, the so called *Expression Problem* [27]. The expression problem is a trade off between a deep and shallow embedding. In a deep embedding, it is easy to add multiple interpretations to the DSL - just add a new evaluation function. However, it is not easy to add a new constructor, since all functions will need to be modified to add a new case for the constructor. The opposite is true in a shallow embedding.

One possible attempt at fixing the expression problem is *Data types à la carte* [25]. It combines constructors using the co-product of their signatures. This technique makes use of standard functors, however, an approach using higher-order functors is described in *Compositional data types* [2].

This is defined as:

```
data (iF :+: iG) f a = L (iF f a) | R (iG f a)
```

It is also the case that if both f and g are IFunctors then so is the sum f :+: g.

```
instance (IFunctor iF, IFunctor iG) ⇒ IFunctor (iF :+: iG) where
  imap f (L x) = L (imap f x)
  imap f (R y) = R (imap f y)
```

For each constructor it is possible to define a new data type and a Functor instance specifying where is recurses. This allows for the modularisation of the parser example:

```
data SatisfyF₂ f a where
  SatisfyF₂ :: (Char → Bool) → SatisfyF₂ f Char
data OrF₂ f a where
  OrF₂ :: f a → f a → OrF₂ f a
instance IFunctor SatisfyF₂ where
  imap f (SatisfyF₂ g) = SatisfyF₂ g
instance IFunctor OrF₂ where
  imap f (OrF₂ px py) = OrF₂ (f px) (f py)
```

By using IFix to tie the recursive knot, the IFix ($\mathsf{SatisfyF_2}$:+: $\mathsf{OrF_2}$) data type would be isomorphic to the original $\mathsf{Parser_d}$ datatype found in Section 2.2.1.
One problem that now exists, however, is that it is now rather difficult to create expressions. Revisiting the simple example of a parser for `'a'` or `'b'`.

```
exampleParser :: IFix (SatisfyF₂ :+: OrF₂) Char
exampleParser = IIn (R (OrF₂ (IIn (L (SatisfyF₂ (≡ 'a')))) (IIn (L (SatisfyF₂ (≡ 'b'))))))
```

It would be beneficial if there was a way to add these Ls and Rs automatically. Fortunately, there is a method using injections. The :≺: type class captures the notion of subtypes between IFunctors.

```
class (IFunctor iF, IFunctor iG) ⇒ iF :≺: iG where
  inj :: iF f a → iG f a
instance IFunctor iF ⇒ iF :≺: iF where
  inj = id
instance (IFunctor iF, IFunctor iG) ⇒ iF :≺: (iF :+: iG) where
  inj = L
instance (IFunctor iF, IFunctor iG, IFunctor iH, iF :≺: iG) ⇒ iF :≺: (iH :+: iG) where
  inj = R · inj
```

Using this type class, smart constructors are defined:

```
inject :: (iG :≺: iF) ⇒ iG (IFix iF) a → IFix iF a
inject = IIn · inj
satisfy₂ :: (SatisfyF₂ :≺: iF) ⇒ (Char → Bool) → IFix iF Char
satisfy₂ f = inject (SatisfyF₂ f)
or₂ :: (OrF₂ :≺: iF) ⇒ IFix iF a → IFix iF a → IFix iF a
or₂ px py = inject (OrF₂ px py)
```

Expressions can now be built using the constructors, such as $\mathsf{satisfy_2}$ (≡ `'a'`) \`$\mathsf{or_2}$\` $\mathsf{satisfy_2}$ (≡ `'b'`).
A modular algebra can now be defined that provides an interpretation of this datatype.

```
newtype Size a = Size {unSize :: Int} deriving Num
class IFunctor iF ⇒ SizeAlg iF where
  sizeAlg :: iF Size a → Size a
instance (SizeAlg iF, SizeAlg iG) ⇒ SizeAlg (iF :+: iG) where
  sizeAlg (L x) = sizeAlg x
  sizeAlg (R y) = sizeAlg y
instance SizeAlg OrF₂ where
  sizeAlg (OrF₂ px py) = px + py
```

```
instance SizeAlg SatisfyF2 where
  sizeAlg (SatisfyF2 f) = 1
eval :: SizeAlg iF ⇒ IFix iF a → Size a
eval = icata sizeAlg
```

The main benefit of this approach is modularity. Each constructor is given by its interpretation in isolation and only for interpretations that make sense for it. Additionally, existing interpretations are not affected by the addition of new constructors, such as ApF_2. This helps to solve the expression problem.

2.5 Dependently Typed Programming

Although Haskell does not officially support dependently typed programming, there are techniques available that together can be used to replicate some of the experience.

2.5.1 DataKinds Language Extension

Through the use of the DataKinds language extension [30], all data types can be promoted to also be kinds and their constructors to be type constructors. When constructors are promoted to type constructors, they are prefixed with a ′. This allows for more interesting and restrictive types.

Consider the example of a vector that also maintains its length. Peano numbers can be used to keep track of the length, which prevents a negative length for a vector. This is where numbers are defined as zero or a number n incremented by 1.

```
data Nat = Zero
  | Succ Nat
```

A vector type can now be defined that makes use of the promoted Nat kind.

```
data Vec :: Type → Nat → Type where
  Nil  :: Vec a 'Zero
  Cons :: a → Vec a n → Vec a ('Succ n)
```

The use of DataKinds can enforce stronger types. For example a function can now require that a specific length of vector is given as an argument. With standard lists, this would not be possible, which could result in run-time errors when the incorrect length is used. For example, getting the head of a list. Getting the head of an empty list an error will be thrown. For a vector, a safeHead function can be defined that will not type check if the vector is empty.

```
safeHead :: Vec a ('Succ n) → a
safeHead (Cons x _) = x
```

2.5.2 Singletons

DataKinds are useful for adding extra information back into the types, but how can information be recovered from the types? For example, could a function that gets the length of a vector be defined?

```
vecLength :: Vec a n → Nat
```

This is enabled through the use of singletons [8]. A singleton in Haskell is a type that has just one inhabitant. That is that there is only one possible value for each type. They are written in such a way that pattern matching reveals the type parameter. For example, the corresponding singleton instance for Nat is SNat. The structure for SNat closely flows that of Nat.

```
data SNat (n :: Nat) where
  SZero :: SNat 'Zero
  SSucc :: SNat n → SNat ('Succ n)
```

A function that fetches the length of a vector can now definable.

```
vecLength2 :: Vec a n → SNat n
vecLength2 Nil          = SZero
vecLength2 (Cons x xs) = SSucc (vecLength2 xs)
```

Recovering an SNat Although being able to define a function that can recover the length of a vector is great, there is a more general way this can be approached. This is to define a new type class that is able to recover an SNat from any type level Nat:

```
class IsNat (n :: Nat) where
  nat :: SNat n
```

The type class has one value inside it nat, which can produce an SNat for a type level Nat. There are two instances from this type class: a base case and a recursive case.

```
instance IsNat 'Zero where
  nat = SZero
instance IsNat n ⇒ IsNat ('Succ n) where
  nat = SSucc nat
```

The base case matches on the type level Nat 'Zero, in this case nat is defined to be SZero — the singleton equivalent. The recursive step deals with the 'Succ n case, where the singleton equivalent SSucc is used to define nat.

A new vector length function can then be defined as:

```
vecLength3 :: IsNat n ⇒ Vec a n → SNat n
vecLength3 _ = nat
```

2.5.3 Type Families

Now consider the possible scenario of appending two vectors together. How would the type signature look? This leads to the problem where two type-level Nats need to be added together. This is where Type Families [21] become useful, they allow for the definition of functions on types. Consider the example of appending two vectors together, this would require type-level arithmetic — adding the lengths together.

```
vecAppend :: Vec a n → Vec a m → Vec a (n :+ m)
```

This requires a :+ type family that can add two Nats together.

2.5.4 Heterogeneous Lists

Heterogeneous lists [14] are a way of having multiple types in the same list. Rather than be parameterised by a single type, they instead make use of a type list, which is the list type promoted through DataKinds to be a kind, with its elements being types. Each element in the type list aligns with the value at that position in the list, giving its type. A heterogeneous list is defined as:

```
data HList (xs :: [Type]) where
  HNil  :: HList '[]
  HCons :: x → HList xs → HList (x ': xs)
```

This data type has two constructors:

- HNil represents the empty list. The type parameter is the empty type list '[]
- HCons allows a new element to be added to the list. The type parameter is the type of the item inserted consed onto the front of the types of the tail of the list.

Functions on HLists

Length using singletons and type families, it is possible to get the length of a HList in a type-safe way. Firstly, a type family is defined that is able to return the length of a type list.

```
type family Length (l :: [k]) :: Nat where
  Length '[]      = 'Zero
  Length (e ': l) = 'Succ (Length l)
```

Length follows a similar definition to the length :: [a] → Int function defined in the Prelude. The base case of Length defines the length to be 'Zero. The recursive case increments the length by 1 for each item in the list, until it reaches the base case.

Now a function is defined that returns the length of a HList:

```
lengthH :: HList xs → SNat (Length xs)
lengthH HNil           = SZero
lengthH (HCons _ xs) = SSucc (lengthH xs)
```

This follows the same structure as the Length type family, however instead, of working with types it uses singleton values.

Take Another function that may be helpful with HLists is take. This will return the first n items from the list. If n is larger than the length of the list, then the whole list will be returned. Again, to be able to do this a new type family is needed – Take:

```
type family Take (n :: Nat) (l :: [k]) :: [k] where
  Take 'Zero      l          = '[]
  Take ('Succ n) '[]        = '[]
  Take ('Succ n) (e ': l) = e ': Take n l
```

The type family follows the same definition as the standard take :: Int → [a] → [a] as defined in the Prelude. Similar to the lengthH function, takeH follows the same structure as the type family:

```
takeH :: SNat n → HList xs → HList (Take n xs)
takeH SZero      l                = HNil
takeH (SSucc n) HNil            = HNil
takeH (SSucc n) (HCons x xs) = HCons x (takeH n xs)
```

Drop The final function used in this project on Hlists is one that can drop the first n elements. The Drop type family can be defined as:

```
type family Drop (n :: Nat) (l :: [k]) :: [k] where
  Drop 'Zero      l          = l
  Drop ('Succ _) '[]        =' []
  Drop ('Succ n) (_ ': l) = Drop n l
```

The Drop type family also closely follows the definition of drop :: Int → [a] → [a] from the Prelude, and its result is reflected in the values level just as with lengthH and takeH.

```
dropH :: SNat n → HList xs → HList (Drop n xs)
dropH SZero l = l
dropH (SSucc _) HNil = HNil
dropH (SSucc n) (HCons _ xs) = dropH n xs
```

```
type family (a :: Nat) :+(b :: Nat) where
  a :+'Zero   = a
  a :+'Succ b = 'Succ (a :+b)
```

2.5.5 Summary

Together these features allow for dependently typed programming constructs in Haskell:

- DataKinds allow for values to be promoted to types
- Singletons allow types to be demoted to values
- Type Families can be used to define functions that manipulate types.

2.6 Existential Types

Typically, when defining a data type in Haskell, every type variable that exists on the right hand side of the equals, must also be on the left hand side. For example, this is not allowed:

$\mathbf{newtype}\ \mathsf{Bad} = \mathsf{Bad}\ \mathsf{a}$

Existential types [18] are a way to allow this to happen, for example:

$\mathbf{data}\ \mathsf{Good} = \forall \mathsf{a}.\mathsf{Good}\ \mathsf{a}$

One benefit of existential types is that the type variable no longer needs to be on the left hand side of the equals.

There is however, one problem with this approach, the variable will be a random unknown type. To avoid this problem constraints are typically added to the signature, so that there can be a set of functions that work with that variable. For example, the variable could have the Show constraint, so that we are able to use the show function with it:

$\mathbf{data}\ \mathsf{Showy} = \forall \mathsf{a}.\mathsf{Show}\ \mathsf{a} \Rightarrow \mathsf{Showy}\ \mathsf{a}$

It would now be possible to build a list of items that can use the show function.

```
showList :: [Showy]
showList = [Showy 123, Showy "abc"]
```

This list can now store any value with a Show instance defined, by wrapping it in the Showy constructor.

2.7 Phantom Type Parameters

Phantom type parameters [4] could be considered the opposite of existential types. This is when a type variable only appears on the left hand side of the equals. The most basic example is Const, it has two type arguments, but only a is used on the right hand side:

$\mathbf{newtype}\ \mathsf{Const}\ \mathsf{a}\ \mathsf{b} = \mathsf{Const}\ \mathsf{a}$

Phantom type parameters can be used to store information in the types, which can act as further static constraints on the types. Consider an example revolving around locking doors: it should not be possible to lock a door that is open, first it has to be closed and then it can be locked. The state of the door can be represented by a data type that is promoted to a kind with the DataKinds extension. A door can be represented as a type with a phantom type variable, with kind DoorState that record the state of the door:

```
data DoorState = Open | Closed | Locked
data Door (state :: DoorState) where
  Door :: Door state
```

It would then be possible to define functions that can close and lock doors:

```
closeDoor :: Door 'Open → Door 'Closed
lockDoor :: Door 'Closed → Door 'Locked
```

The closeDoor function, enforces that only an open door can be given as input, similarly lockDoor prevents an open door from being locked.

2.8 Monoidal Resource Theories

Resource theories [5] are a branch of mathematics that allow for the reasoning of questions surrounding resources, for example:

- If I have some resources, can I make something?
- If I have some resources, how can I get what I want?

Resource theories provide a way to answer these questions.

2.8.1 Preorders

A preorder relation on a set X is denoted by $\leq$. The relation must obey two laws:

1. Reflexivity — $x \leq x$
2. Transitivity — if $x \leq y$ and $y \leq z$ then $x \leq z$.

A preorder is a pair $(X, \leq)$ made up of a set and a preorder relation on that set.

2.8.2 Symmetric Monoidal Preorders

A symmetric monoidal structure $(X, \leq, I, \otimes)$ on a preorder $(X, \leq)$ has two additional components:

1. The monoidal unit — an element $I \in X$
2. The monoidal product — a function $\otimes : X \times X \to X$

To be a symmetric monoidal then the following laws must be satisfied:

1. Monotonicity — $\forall x_1, x_2, y_1, y_2 \in X$, if $x_1 \leq y_1$ and $x_2 \leq y_2$, then $x_1 \otimes x_2 \leq y_1 \otimes y_2$
2. Unitality — $\forall x \in X$, $I \otimes x = x$ and $x \otimes I = x$
3. Associativity — $\forall x, y, z \in X$, $(x \otimes y) \otimes z = x \otimes (y \otimes z)$
4. Symmetry — $\forall x, y \in X$, $x \otimes y = y \otimes x$

2.8.3 Wiring Diagrams

A wiring diagram is made up of: boxes that can have multiple inputs and outputs. The boxes can be arranged in series or in parallel. Figure 2.5, shows an example wiring diagram.

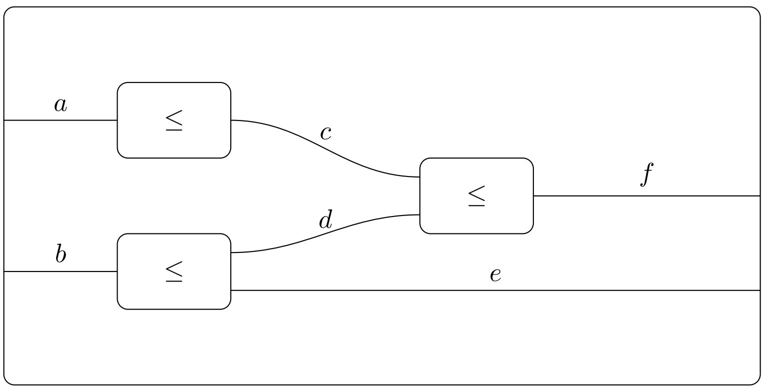
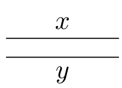


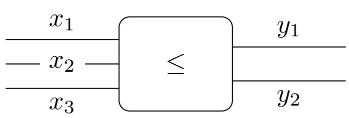
Figure 2.5: An example wiring diagram

A wiring diagram can be formalised as a symmetric monoidal preorder. Each element $x \in X$ can exists as the label on a wire. Two wires x and y, can be drawn in parallel:



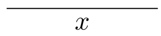
Two wires in parallel are be considered to be the monoidal product $x \otimes y$. The monodial unit is defined as a wire with the label I or no wire.

A box connects parallel wires on the left to parallel wires on the right. A wiring diagram is considered valid if the monoidal product of the left is less than the right.

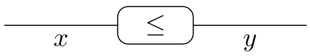


This example wiring diagram corresponds to the inequality $x_1 \otimes x_2 \otimes x_3 \leq y_1 \otimes y_2$.

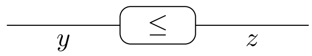
Reflexivity The reflexivity law states that $x \leq x$, this states that a diagram of one wire is valid.



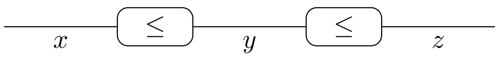
Transitivity The transitivity law says that if $x \leq y$ and $y \leq z$ then $x \leq y$. This corresponds to connecting two diagrams together in sequence. If both of the diagrams



and

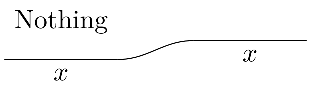


are valid, then they can be joined together to obtain another valid diagram.



Monotonicity Monotonicity states that, if $x_1 \leq y_1$ and $x_2 \leq y_2$, then $x_1 \otimes x_2 \leq y_1 \otimes y_2$. This can be thought of as stacking two boxes on top of each other:

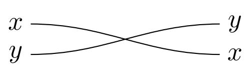
Unitality The unitality law states that $I \otimes x = x$ and $x \otimes I = x$, this means that a blank space can be ignored and that diagrams such as



are valid.

Associativity The associativity law says that $(x \otimes y) \otimes z = x \otimes (y \otimes z)$, this states that diagrams can be built from either the top or bottom. In reality it is trivial to see how this is true with wires:

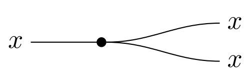
Symmetry The symmetry law states that $x \otimes y = y \otimes x$, this encodes the notion that a diagram is still valid even if the wires cross.



Discard Axiom There are times when there is no longer need to keep a value, it would be beneficial if it could be discarded. In a wiring diagram this is represented as:

This can be added as an additional axiom to the definition of a symmetric monoidal preorder: $\forall x \in X, x \leq I$

Copy Axiom The final axiom to add is the notion of copying a value: $\forall x \in X, x \leq x + x$. This can be represented in wiring diagram as a split wire:



Chapter 3

The Language

3.1 Language Requirements

For the design of the language to be considered a success, several criteria need to be met:

- **Describe any dataflow** — The combinators in the language should be able to describe any dataflow that the user needs.
- **Quickness to Learn** — The language should be quick and easy to learn. If it takes users too long to learn, they may never bother, meaning that the language will never be used.
- **Easy to write** — A user should be able to write programs easily. They should not have to spend time creating additional boilerplate code, where it is not necessary.
- **Easy to understand** — Any program written with this language should be easy to understand, so that users can review existing code and know what it will do.
- **Type-safe** — A feature missing in many dataflow tools is the lack of type checking. This causes problems later on in the development process with more debugging and testing needed. The language should be type-safe to avoid any run-time errors occurring where types do not match.

3.2 Tasks

Dataflow programming focuses on the transforming inputs to outputs. To be able to transform inputs this language will make use of tasks. They are responsible for reading from an input data source, completing some operation on the input, then finally writing to an output data sink. Tasks could take many different forms, for example they could be:

- A pure function — a function with type $\mathsf{a} \to \mathsf{b}$
- An external operation — interacting with some external system. For example, calling a terminal command.

A task could have a single input or multiple inputs, however, for now just a single input task will be considered. Multi-input tasks are explained further in Sub-Section 3.4.3

3.2.1 Data Stores

Data stores are used to pass values between different tasks, this ensures that the input and output of tasks are closely controlled. They are used to represent different ways of storing values: one example could be a point to a CSV file. By also having just one place that defines how to read and write to data stores, it will reduce the possibility of an error occurring and make it easier to test. A data store can be defined as a type class, with two methods fetch and save:

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

A DataStore has two type parameters: where f is the type of DataStore being used and a is the type of the value stored inside it. The aptly named methods describe their intended function: fetch will fetch a value from a DataStore, and save will save a value. The fetch method takes a DataStore as input and will return the value stores inside. However, the save method may not be as self explanatory, since it has an extra f a argument. This argument can be thought of as a pointer to a DataStore: it contains the information needed to save. For example, in the case of a file store it could be the file name.

By implementing this as a type class, there can be many different implementations of a DataStore. The library comes with several pre-defined DataStores, such as a VariableStore. This can be though of as an in memory storage between tasks.

```
data VariableStore a = Var a | Empty
instance DataStore VariableStore a where
  fetch (Var x) = return x
  save Empty x = return (Var x)
```

The VariableStore is the most basic example of a DataStore, a more complex example is a FileStore, which represents a pointer to a file:

```
newtype FileStore a = FileStore String
instance DataStore FileStore String where
  fetch (FileStore fname) = readFile fname
  save (FileStore fname) x = writeFile fname x >> return (FileStore fname)
instance DataStore FileStore [String] where
  fetch (FileStore fname) = readFile fname >>= return · lines
  save (FileStore fname) x = writeFile fname (unlines x) >> return (FileStore fname)
```

The FileStore is only defined to store two different types: String and [String]. If a user attempts to store anything other than these two types then a compiler error will be thrown, for example:

```
ghci> save (FileStore "test.txt") (123 :: Int)
> No instance for (DataStore FileStore Int) arising from a use of 'save'
```

Although a small set of DataStores are included in the library, the user is also able to add new instances of the type class with their own DataStores. Some example expansions, could be supporting writing to a database table, or a Hadoop file system.

3.2.2 Task Constructor

The type of a task details the inputs and outputs. A task is created via a constructor that takes two arguments: the function it represents and somewhere to store the output. This constructor makes use of GADTs syntax [20] so that constraints can be placed on the types used. It enforces that a DataStore must exist for the input and output types. This allows the task to make use of the fetch and save functions.

```
data Task (f :: Type → Type) (a :: Type) (g :: Type → Type) (b :: Type) where
  Task :: (DataStore f a, DataStore g b) ⇒ (f a → g b → IO (g b)) → g b → Task f a g b
```

When a Task is executed the stored function is executed, with the input being passed in as the first argument and the output "pointer" as the second argument. This returns an output DataStore that can be passed on to another Task.

3.3 Chains, A Dalliance

In a dataflow programming, one of the key aspects is the definition of dependencies between tasks in the flow. One possible approach to encoding this concept in the language, that was ultimately not up to scratch, is to make use of sequences of tasks — also referred to as chains. These chains compose tasks, based on their dependencies. A chain can be modelled with an abstract datatype:

```
data Chain (f :: Type → Type) (a :: Type) (g :: Type → Type) (b :: Type) where
  Chain :: Task  f a g b → Chain f  a g b
  Then  :: Chain f a g b → Chain g b h c → Chain f a h c
```

The Chain constructor wraps a Task in the Chain data type. This allows for them to be easily composed with other Tasks. Without this there would need to be an “empty” element, however, this could lead to the construction of a chain with no tasks. The Then constructor combines multiple chains to form a sequence of sequential tasks. To make this easier to use an operator ⋙ is defined that represents Then:

```
(⋙) :: Chain f a g b → Chain g b h c → Chain f a h c
(⋙) = Then
```

This can be now be used to join sequences of tasks together, for example:

```
task1 :: Task VariableStore Int       VariableStore String
task2 :: Task VariableStore String    FileStore     [String]
task3 :: Task FileStore     [String]  VariableStore Int

sequence :: Chain VariableStore Int VariableStore Int
sequence = Chain task1 ⋙ Chain task2 ⋙ Chain task3
```

sequence will perform the three tasks in order, starting with task1 and finishing with task3.

3.3.1 Trees as Chains

Now that tasks can be performed in sequence, the next logical step will be to introduce the concept of branching out. This results in a tasks output being given to multiple tasks, rather than just 1.

To do this a new abstract datatype is required. This will be used to form a list of Chains, conventionally the [] type would be used, however this is not possible as each chain will have a different type. This means that existential types will need to be used.

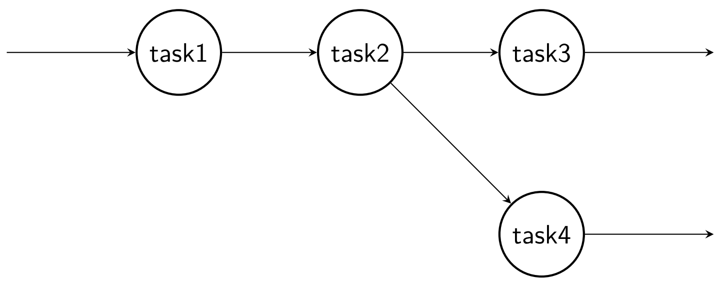
```
data Pipe where
  Pipe :: ∀f a g b.(DataStore f a, DataStore g b) ⇒ Chain f a g b → Pipe
  And  :: Pipe → Pipe → Pipe
```

The And constructor can be used to combine multiple chains together. Figure 3.1 shows the previous sequence, with a new task4 which also uses the input from task2.

```
task4 :: Task FileStore [String] VariableStore String

branchExample :: Pipe
branchExample = Pipe (Chain task1 ⋙ Chain task2 ⋙ Chain task3)
                `And`
                Pipe (Chain task2 ⋙ Chain task4)
```

(a)



(b)

Figure 3.1: A Pipe (a) and its corresponding dataflow diagram (b).

However, there is a problem with this approach: to be able to form a network similar to that shown in Figure 3.1, the language will need to know where to join two Chains together. However, with the current definition of a Task, it is not possible to easily check the equivalence of two functions. Being able to check for equivalence will be key to defining a method to merge these chains together: it will need to match task2 in the first chain with the task2 in the second chain. Similarly, if a user wanted to use the same task multiple times, it would not be possible to differentiate between them. One approach to this would be to have unique identifiers for each task, such as PIDs.

Process Identifiers (PIDs) A Chain can be modified so that instead of storing a Task it instead stores a PID — a unique identifier for a task. However to do this a new PID data type is needed:

```
data PID (f :: Type → Type) (a :: Type) (g :: Type → Type) (b :: Type) where
  PID :: Int → PID f a g b
```

The PID data type has the same kind as a Task and makes use of phantom type parameters, to retain the same type information as a Task, whilst storing just an Int that can be used to identify it.

```
data Chain′ (f :: Type → Type) (a :: Type) (g :: Type → Type) (b :: Type) where
  Chain′ :: PID    f a g b → Chain′ f a g b
  Then′  :: Chain′ f a g b → Chain′ g b h c → Chain′ f a h c
```

This new version of Chain named Chain′, is almost identical in the way it behaves, however instead of storing a Task, it stores a PID.

This, however, leaves a key question, how do Tasks get mapped to PIDs. This can be done by employing the State monad. The state stores a map from PID to task and a counter for PIDs. As Tasks each have a different type a new wrapper datatype is required, making use of existential types, to close over the types, and produce values of apparently the same type. This is because a Map can only store one type. The Workflow monad is a type alias for the State monad, which stores the WorkflowState.

```
data TaskWrap = ∀f a g b.TaskWrap (Task f a g b)
data WorkflowState = WorkflowState {
  pidCounter :: Int,
  tasks :: M.Map Int TaskWrap
}
type Workflow = State WorkflowState
```

An operation that is defined for the monad is registerTask. This takes a Task and returns a Chain that stores a PID inside it. Whenever a user would like to add a new task to the workflow, they register it. They can then use this returned value to construct multiple chains, which can now be joined easily by comparing the stored PID.

```
registerTask :: Task f a g b → Workflow (Chain′ f a g b)
```

One benefit to this approach is that if the user would like to use a task again in a different place, they can simply register it again and use the new PID value.

3.3.2 Evaluation

try fix the half if i have time

☒ **Describe any Dataflow** This method would be capable of describing any dataflow, although in its current state, it can only support trees. The algorithm that would be used to join different chains together could be developed to allow them to rejoin onto another chain. This will allow for any Directed Acyclic Graph (DAG) to be defined.

☑ **Quickness to Learn** With only 4 different combinators (Chain, ⋙, Pipe, And) and the Task constructor, this language should be simple to learn. There are only two main concepts the user needs to understand: how to join tasks into a chain and how to join chains together.

☑ **Easy to Write** Building chains is a very simple process, and allows the user to focus on the dependencies of one task. They do not need to be aware of the bigger picture. For example, adding a new task5 that depends on task2 in Figure 3.1. The user will just need to add one extra chain task5 ⋙ task2, which will have no effect on the existing chains.

☒ **Easy to Read** Although it is easy to write chains, this could lead to messy definitions with no structure. This will make it harder for a user to interpret an already defined collection of chains. Its possible that they will need to draw out some of the dependencies to further understand what is already defined.

☒ **Type-safe** Although a Chain can be well typed, the use of existential types to join chains together pose a problem. This causes the types to be 'hidden', this means that when executing these tasks, the types need to be recovered. This is possible through the use of gcast from the Data.Typeable library. However, this has to perform a reflection at run-time to compare the types. There is the possibility that the types could not matching and this would only be discovered at run-time. There is a mechanism to handle the failed match case, however, this does not fulfil the criteria of being fully type-safe.

Chains are not good enough Chains do not satisfy some of the key requirements that this library set out to solve, such as type safety. This could lead to crashes at run-time as the compiler is not able to fully verify that the system type-checks. Another approach could be to look to category theory, for ways to compose functions together in a type-safe way.

3.4 Solution: Circuit

This approach is inspired by monadic resource theory, which has a collection of mathematical operators for composing functions together. It uses parallel prefix circuits, described by Hinze [10], as a starting point for the design of the combinators. A set of constructors can be defined that are used to represent a DAG. The constructors seen in Figure 3.2 represent the behaviour of edges in the graph.

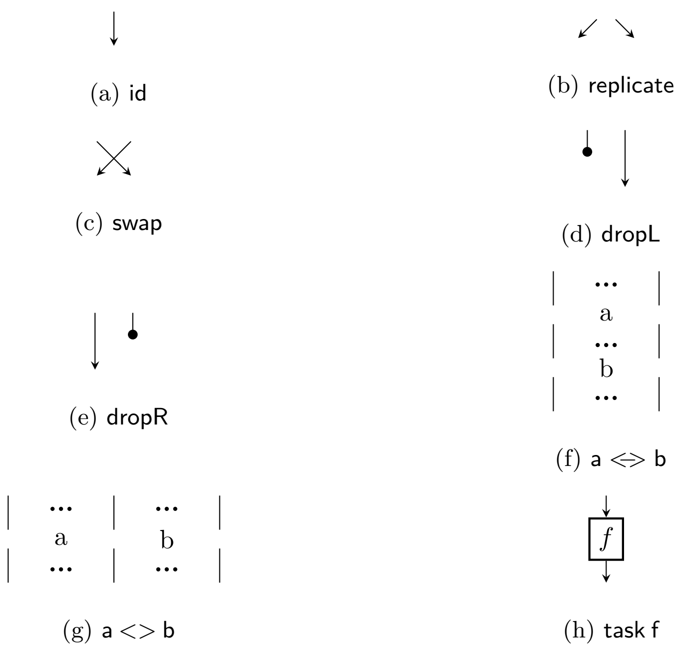


Figure 3.2: The constructors in the Circuit library alongside their graphical representation.

3.4.1 Constructors

Each of these constructors use strong types to ensure that they are combined correctly. A Circuit has 7 different type parameters:

$$\begin{aligned}&\mathsf{Circuit}\ (\mathsf{inputsStorageTypes}\ :: [\mathsf{Type} \to \mathsf{Type}])\ (\mathsf{inputsTypes}\ :: [\mathsf{Type}])\ (\mathsf{inputsApplied}\ :: [\mathsf{Type}])\\ &\quad (\mathsf{outputsStorageTypes} :: [\mathsf{Type} \to \mathsf{Type}])\ (\mathsf{outputsTypes} :: [\mathsf{Type}])\ (\mathsf{outputsApplied} :: [\mathsf{Type}])\\ &\quad (\mathsf{nInputs} :: \mathsf{Nat})\end{aligned}$$

A Circuit can be thought of as a list of inputs, which are processed and a resulting list of outputs are produced. To represent this it makes use of the DataKinds language extension [30], to use type-level lists and natural numbers. Each parameter represents a certain piece of information needed to construct a circuit:

- inputsStorageTypes is a type-list of storage types, for example '[VariableStore, CSVStore] — these all have kind $* \to *$.

- inputsTypes is a type-list of the types stored in the storage, for example '[Int, [(String, Float)]].
- inputsApplied is a type-list of the storage types applied to the types stored, for example '[VariableStore Int, CSVStore [(String, Float)]].
- outputsStorageTypes, outputsTypes and outputsApplied mirror the examples above, but for the outputs instead.
- nInputs is a type-level Nat that is the length of the input lists.

Although, to a human some of these types may seem irrelevant, GHC is not able to make all of the deductions itself, when type checking the code. It requires additional information, such as inputsApplied or nInputs.

In the language there are two different types of constructor, those that recurse and those that can be considered leaf nodes. The behaviour of both types of constructor is recorded within the types. For example, the id constructor has the type:

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

It can be seen how the type information for this constructor states that it has 1 input value (N1∼'Succ'Zero) of type f a and it returns that same value. Each type parameter in id is a phantom type [4], since there are no values stored in the data type that use the type parameters. The rest of the constructors that are leaf nodes are: replicate, swap, dropL, and dropR:

```
replicate :: DataStore' '[f]    '[a]    ⇒ Circuit '[f]    '[a]    '[f a]      '[f, f] '[a, a] '[f a, f a] N1
swap      :: DataStore' '[f, g] '[a, b] ⇒ Circuit '[f, g] '[a, b] '[f a, g b] '[g, f] '[b, a] '[g b, f a] N2
dropL     :: DataStore' '[f, g] '[a, b] ⇒ Circuit '[f, g] '[a, b] '[f a, g b] '[g]    '[b]    '[g b]      N2
dropR     :: DataStore' '[f, g] '[a, b] ⇒ Circuit '[f, g] '[a, b] '[f a, g b] '[f]    '[a]    '[f a]      N2
```

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. The swap constructor takes two values as input: f a and g b. It will then swap these values over, such that the output will now be: g b and f a. dropL will take two inputs: f a and g b. It will then drop the left argument and return just a g b. The dropR has the same behaviour as dropL, it just drops the right argument instead.

To be able to make use of the leaf nodes, they need to be combined in some way. To do this two new recursive constructors named 'beside' and 'then' will be used. However, before defining these constructors there are some tools that are required. This is due to the types no longer being concrete. For example, the input type list is no longer known: it can only be referred to as fs and as. This means it is much harder to specify the new type of the Circuit.

Apply Type Family It would not be possible to use a new type variable xs for the inputsApplied parameter. This is because it needs to be constrained so that it is equivalent to fs applied to as. To solve this a new closed type family [7] is created that is able to apply the two type lists together. This type family pairwise applies a list of types with kind $* \to *$ to a list of types with kind $*$ to form a new list containing types of kind $*$. For example, Apply '[f, g, h] '[a, b, c]∼'[f a, g b, h c].

': looks awful

```
type family Apply (fs :: [Type → Type]) (as :: [Type]) where
  Apply '[] '[] = '[]
  Apply (f ': fs) (a ': as) = f a ': Apply fs as
```

Append Type Family There will also be the need to append two type level lists together. Lists would need to be appended in this way, when combining inputs and outputs to form a larger circuit. For example, '[a, b, c] :++ '[d, e, f]∼'[a, b, c, d, e, f]. To do this an append type family [14] can be used:

```
type family (:++) (l1 :: [k]) (l2 :: [k]) :: [k] where
  (:++) '[]       l  = l
  (:++) (e ': l) l' = e ': (l :++ l')
```

The :++ type family is defined in the same way as the standard ++ function on value lists, however, it appends type lists together instead. This type family makes use of the language extension PolyKinds [30] to allow for the append to be polymorphic on the kind stored in the type list. This will avoid defining multiple versions to append fs with gs, and as with bs.

The 'Then' Constructor This constructor — denoted by <=> — is used to stack two circuits on top of each other. It is used to encapsulate the idea of dependencies, between different circuits. Through types it enforces that the output of the top circuit is the same as the input to the bottom circuit.

$$\begin{aligned}
&(<\!\!=\!\!>) :: (\mathsf{DataStore'}\ \mathsf{fs}\ \mathsf{as}, \mathsf{DataStore'}\ \mathsf{gs}\ \mathsf{bs}, \mathsf{DataStore'}\ \mathsf{hs}\ \mathsf{cs})\\
&\quad \Rightarrow \mathsf{Circuit}\ \mathsf{fs}\ \ \mathsf{as}\ (\mathsf{Apply}\ \mathsf{fs}\ \ \mathsf{as})\ \mathsf{gs}\ \mathsf{bs}\ (\mathsf{Apply}\ \mathsf{gs}\ \mathsf{bs})\ \mathsf{nfs}\\
&\quad \to \mathsf{Circuit}\ \mathsf{gs}\ \mathsf{bs}\ (\mathsf{Apply}\ \mathsf{gs}\ \mathsf{bs})\ \mathsf{hs}\ \mathsf{cs}\ (\mathsf{Apply}\ \mathsf{hs}\ \mathsf{cs})\ \mathsf{ngs}\\
&\quad \to \mathsf{Circuit}\ \mathsf{fs}\ \ \mathsf{as}\ (\mathsf{Apply}\ \mathsf{fs}\ \ \mathsf{as})\ \mathsf{hs}\ \mathsf{cs}\ (\mathsf{Apply}\ \mathsf{hs}\ \mathsf{cs})\ \mathsf{nfs}
\end{aligned}$$

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

The 'Beside' Constructor Denoted by <>, the beside constructor is used to place two circuits side-by-side. The resulting Circuit has the types of left and right circuits appended together.

$$\begin{aligned}
&(<>) :: (\mathsf{DataStore'}\ \mathsf{fs}\ \mathsf{as}, \mathsf{DataStore'}\ \mathsf{gs}\ \mathsf{bs}, \mathsf{DataStore'}\ \mathsf{hs}\ \mathsf{cs}, \mathsf{DataStore'}\ \mathsf{is}\ \mathsf{ds})\\
&\quad \Rightarrow \mathsf{Circuit}\ \mathsf{fs}\ \ \mathsf{as}\ (\mathsf{Apply}\ \mathsf{fs}\ \ \mathsf{as})\ \mathsf{gs}\ \mathsf{bs}\ (\mathsf{Apply}\ \mathsf{gs}\ \mathsf{bs})\ \mathsf{nfs}\\
&\quad \to \mathsf{Circuit}\ \mathsf{hs}\ \mathsf{cs}\ (\mathsf{Apply}\ \mathsf{hs}\ \mathsf{cs})\ \mathsf{is}\ \ \mathsf{ds}\ (\mathsf{Apply}\ \mathsf{is}\ \ \mathsf{ds})\ \mathsf{nhs}\\
&\quad \to \mathsf{Circuit}\ (\mathsf{fs}\ \ {:}{+\!\!+}\ \mathsf{hs})\ \ (\mathsf{as}\ {:}{+\!\!+}\ \mathsf{cs})\ (\mathsf{Apply}\ \mathsf{fs}\ \ \mathsf{as}\ {:}{+\!\!+}\ \mathsf{Apply}\ \mathsf{hs}\ \mathsf{cs})\\
&\qquad\qquad\quad (\mathsf{gs}\ {:}{+\!\!+}\ \mathsf{is})\ \ \ (\mathsf{bs}\ {:}{+\!\!+}\ \mathsf{ds})\ (\mathsf{Apply}\ \mathsf{gs}\ \mathsf{bs}\ {:}{+\!\!+}\ \mathsf{Apply}\ \mathsf{is}\ \ \mathsf{ds})\\
&\qquad\qquad\quad (\mathsf{nfs}\ {:}{+}\mathsf{nhs})
\end{aligned}$$

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 — defined in Section 2.5.3 — to add the number of inputs from the left and right together.

3.4.2 Combined Data Stores

A keen eyed reader may notice that all of these constructors have not been using the original DataStore type class. Instead they have all used the DataStore′ type class. This is a special case of a DataStore, allowing for constructors to also be defined over type lists, not just a single type. Combined data stores make it easier for tasks to fetch from multiple inputs. Users will just have to call a single fetch′ function, rather than multiple. However, since tasks can only have one output, there is no need for a save′ function, that would be able to save to multiple data stores.

To be able to define DataStore′, heterogeneous lists [14] are needed — specifically two different forms. HList′ stores values of type f a and is parameterised by two type lists fs and as. IOList stores items of type IO a and is parameterised by a type list as. Using an IOList makes it easier to define a function that produces a list of IO computations. Their definitions are:

```
data HList' (fs :: [Type → Type]) (as :: [Type]) where
  HCons' :: f a → HList' fs as → HList' (f ' : fs) (a ' : as)
  HNil'  :: HList' '[] '[]
data IOList (xs :: [Type]) where
  IOCons :: IO x → IOList xs → IOList (x ' : xs)
  IONil  :: IOList '[]
```

Now that there is a mechanism to represent a list of different types, it is possible to define DataStore′:

```
class DataStore' (fs :: [Type → Type]) (as :: [Type]) where
  fetch' :: HList' fs as → IOList as
```

To save the user of the cumbersome task of having to define an instance of DataStore′ for every possible combination of data stores, the instance is derived from the previous DataStore type class. This means that a user does not need to create any instances of DataStore′. They can instead focus on each single case, with the knowledge that they will automatically be able to combine them with other data stores.

```
instance {-# OVERLAPPING #-} (DataStore f a) ⇒ DataStore' '[f] '[a] where
  fetch' (HCons' x HNil') = IOCons (fetch x) IONil
```

$$\textbf{instance}\ (\mathsf{DataStore\ f\ a}, \mathsf{DataStore'\ fs\ as}) \Rightarrow \mathsf{DataStore'\ (f\ '\!:\ fs)\ (a\ '\!:\ as)}\ \textbf{where}$$
$$\quad \mathsf{fetch'\ (HCons'\ x\ xs)} \quad = \mathsf{IOCons\ (fetch\ x)\ (fetch'\ xs)}$$

One of these instances makes use of the {-# OVERLAPPING #-} pragma. In most cases the base case instance that would be defined is $\mathsf{DataStore'\ '[\,]\ '[\,]}$. However, it does not makes sense to have an empty data store. Therefore, the base case is selected to be a list with one element $\mathsf{DataStore'\ '[f]\ '[a]}$. This leads to a problem: GHC is unable to decide which instance to use. It could use either the $\mathsf{DataStore'\ '[f]\ '[a]}$ or the $\mathsf{DataStore'\ (f\ ':\ '[\,])\ (a\ ':\ '[\,])}$ instance. The overlapping pragma tells GHC, that if it encounters this scenario, it should choose the one with the pragma.

3.4.3 Multi-Input Tasks

With a Circuit, it is possible to represent a DAG. This means that a node in the graph can now have multiple dependencies, as seen in Figure 3.3.

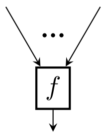


Figure 3.3: A graphical representation of a task with multiple dependencies

To support this, a modification is made to the task constructor: rather than have an input value type of f a, it can now have an input value type of HList′ fs as. The function executed in the task can now use fetch′ to fetch all inputs with one function call.

$$\begin{aligned}&\mathsf{task} :: (\mathsf{DataStore'\ fs\ as}, \mathsf{DataStore\ g\ b})\\&\quad \Rightarrow (\mathsf{HList'\ fs\ as} \to \mathsf{g\ b} \to \mathsf{IO\ (g\ b)})\\&\quad \to \mathsf{g\ b}\\&\quad \to \mathsf{Circuit\ fs\ as\ (Apply\ fs\ as)\ '[g]\ '[b]\ '[g\ b]\ (Length\ fs)}\end{aligned}$$

Now that the length of the inputs is unknown, in order to specify the nInputs type parameter, the Length type family defined in Section 2.5.4 must be used. This will return a type-level Nat, which is the length of the input array fs.

Smart Constructors There could be many times that the flexibility provided by defining your own tasks from scratch could cause a large amount of boiler plate code. For example, there may be times that a user already has pre-defined function and would like to convert it to a task. Therefore there are also two smart constructors that they are able to use:

$$\begin{aligned}&\mathsf{multiInputTask} :: (\mathsf{DataStore'\ fs\ as}, \mathsf{DataStore\ g\ b})\\&\quad \Rightarrow (\mathsf{HList\ as} \to \mathsf{b})\\&\quad \to \mathsf{g\ b}\\&\quad \to \mathsf{Circuit\ fs\ as\ (Apply\ fs\ as)\ '[g]\ '[b]\ '[g\ b]\ (Length\ fs)}\\&\mathsf{functionTask} :: (\mathsf{DataStore\ f\ a}, \mathsf{DataStore\ g\ b})\\&\quad \Rightarrow (\mathsf{a} \to \mathsf{b})\\&\quad \to \mathsf{g\ b}\\&\quad \to \mathsf{Circuit\ '[f]\ '[a]\ '[f\ a]\ '[g]\ '[b]\ '[g\ b]\ N1}\end{aligned}$$

The first allows for a simple function with multiple inputs to be defined. With the fetching and saving handled by the smart constructor. The second allows for a simple $\mathsf{a} \to \mathsf{b}$ function to be turned into a Task.

3.4.4 mapC operator

Currently a circuit has a static design — once it has been 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. This could be combated with more smart constructors to generate more complex circuits, with the pre-existing constructors. Another approach would be to add new constructors that allow for more dynamic circuits, such as mapC. This new constructor is used to map a circuit on a single input containing a list of items. The input is fed into the inner circuit, accumulated back into a list, and then output.

$$
\begin{array}{llllll}
\multicolumn{6}{l}{\mathsf{mapC} :: (\mathsf{DataStore}'\ '[\mathsf{f}]\ '[[\mathsf{a}]], \mathsf{DataStore}\ \mathsf{g}\ [\mathsf{b}])} \\
\quad \Rightarrow \mathsf{Circuit}\ '[\mathsf{VariableStore}]\ '[\mathsf{a}] & '[\mathsf{VariableStore}\ \mathsf{a}] & '[\mathsf{VariableStore}] & '[\mathsf{b}] & '[\mathsf{VariableStore}\ \mathsf{b}] & \mathsf{N1} \\
\quad \rightarrow \mathsf{g}\ [\mathsf{b}] \\
\quad \rightarrow \mathsf{Circuit}\ '[\mathsf{f}] & '[[\mathsf{a}]]\ '[\mathsf{f}\ [\mathsf{a}]] & '[\mathsf{g}] & '[[\mathsf{b}]]\ '[\mathsf{g}\ [\mathsf{b}]] & & \mathsf{N1}
\end{array}
$$

This example can be though of a production line ([a] → [b]). The circuit given as an argument describes how to produce one item (a → b). The mapC can then be provided with a pallet (or a list) of resources to build multiple items ([a]), it will then return a pallet of made items ([b]).

3.4.5 Completeness

The constructors in this library make up a symmetric monoidal preorder. For simplicity only the inputsApplied and outputsApplied type parameters will be used to formalise a Circuit — all other type parameters are only required to aid GHC in compilation.

A preorder is defined over tasks and DataStores. The preorder relation $\leq$, can be used to describe the dependencies in the DataStores, with a task being able to transform DataStores into new DataStores. The relation is defined over the set X, which describes the set of all possible DataStores.

The monoidal product $\otimes$ can be thought of as the concatenation of multiple DataStores into type-lists. For example the monoidal product of $(f\ a) \otimes (g\ b)$ = '[f a] :++' [g b]∼'[f a, g b]. The monoidal unit, is tricky to define as it has no real meaning within a Circuit, however it could be considered the empty DataStore: '[ ].

The axioms are then satisfied as follows:

1. Reflexivity — this is the id :: Circuit '[f a] '[f a] constructor, it represents a straight line with the same input and output.

2. Transitivity — this is the <=> ::Circuit x y → Circuit y z → Circuit x z constructor, it allows for circuits to be placed in sequence.

3. Monotonicity — this is the <> :: Circuit x_1 y_1 → Circuit x_2 y_2 → Circuit (x_1 :++ x_2) (y_1 :++ y_2) constructor. This can place circuits next to each other.

4. Unitality — given the monoidal unit '[ ] and a DataStore xs, then the rules hold true: '[ ] :++ xs∼xs and xs :++' [ ]∼xs.

5. Associativity — given three DataStores: xs, ys, zs. Since concatenation of lists is associative then this rule holds: (xs :++ ys) :++ zs∼xs :++ (ys :++ zs).

6. Symmetry — this is the swap :: Circuit '[f a, g b] '[g b, f a] constructor, it allows for values to swap over.

7. Delete Axiom — this is satisfied by the dropL :: Circuit '[f a, g b] '[g b] and dropR :: Circuit '[f a, g b] '[f a]. Although this does not directly fit with the axiom, it also has to ensure the constraint on a circuit that there must always be 1 output value.

8. Copy Axiom — this is the replicate :: Circuit '[f a] '[f a, f a] constructor. It allows for a DataStore to be duplicated.

By satisfying all the axioms a Circuit is a symmetric monoidal preorder.

3.4.6 Evaluation

☑ **Describe any Dataflow** It is possible to represent a DAG with the constructors in this library. Its completeness has been formalised as a symmetric monoidal preorder.

☑ **Quickness to Learn** The language uses, combinators that also have a visual representation, this makes it easy to quickly understand how they all work. A user can then also benefit from the familiarity of using the host language Haskell.

☑ **Easy to Write** Building a circuit is a more difficult task: to be able to define it the user needs to have an understanding of the shape of the dataflow diagram. Once the user has a sketch for the dataflow they would like to create, translating to a circuit is a much simpler job. This creates more upfront work for the user, however, it is offset by the additional benefits that a circuit brings.

☑ **Easy to Read** Due to the graphical nature of the constructors, it is relatively simple to build up a picture of how an existing circuit works. The user is able to infer the shape of the dataflow diagram easily by working their way down a circuit from top to bottom, and visualising how different tasks are connected. It could also be possible to pretty print a circuit to recreate the dataflow diagram — although this has not been implemented.

☑ **Type-safe** One key benefit that a `Circuit` brings is that constructing them uses strong types. Each constructor encodes its behaviour within the types. This allows the GHC type checker to validate a `Circuit` at compile-time, to ensure that each task is receiving the correct values. This avoids the possibility of crashes are run-time, where types do not match correctly. There is, however, a consequence of this type-safety: the user now needs to add some explicit types on a `Circuit` to help the type checker.

Chapter 4

Implementation

4.1 Requirements

The implementation of the network itself has several requirements that are separate from the language design:

- **Type-safe** — This is a continuation of the previous requirement for the language. It is also important that once the user has built a well-typed Circuit, that the code also continues to be executed in a well-typed environment, to ensure that all inputs and outputs are correctly typed.
- **Parallel** — One of the key benefits that comes from dataflow programming is implicit parallelisation. With this DSL being tailored towards data pipelines, which could be computationally expensive, it should be able to benefit from parallel execution.
- **Competitive Speed** — This library should be able to execute dataflows in a competitive time, with other libraries that already exist.
- **Failure Tolerance** — It is important that if one invocation of a task crashes, it does not crash the whole program. This implementation should be able to gracefully handle errors and propagate them through the circuit.
- **Usable** — The implementation of the library should not break any of the usability of the language design.
- **Maintainable** — It should be easy to maintain the library and add new constructors in the future.

4.2 Circuit AST

The constructors for the language are actually *smart constructors*. They provide a more elegant way to build an AST, which represents the circuit. They give the ability to gain 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.

4.2.1 IFunctor

To build the fixed core AST for a Circuit, indexed functors — also known as IFunctor — are used. IFunctors are, however, only defined to take a single type index — a Circuit needs 7. To do this $\mathsf{IFunctor}_7$ can be defined:

$$\begin{aligned}&\textbf{class}\ \mathsf{IFunctor}_7\ \mathsf{iF}\ \textbf{where}\\&\quad \mathsf{imap}_7 :: (\forall \mathsf{a\,b\,c\,d\,e\,f\,g}.\mathsf{f'\,a\,b\,c\,d\,e\,f\,g} \to \mathsf{g'\,a\,b\,c\,d\,e\,f\,g}) \to \mathsf{iF\,f'\,a\,b\,c\,d\,e\,f\,g} \to \mathsf{iF\,g'\,a\,b\,c\,d\,e\,f\,g}\end{aligned}$$

$\mathsf{IFunctor}_7$ follows a similar structure to a standard IFunctor, it just has 7 type indicies instead. This indexed functor can be used to mark the recursive points of the data types used to construct the Circuit AST.

However, any data type that is converted to use an $\mathsf{IFunctor}_7$, will need a way to tie the recursive knot. A new type called IFix_7 can be used, it will follow a similar pattern to IFix, but with 7 type indicies.

```
newtype IFix7 iF a b c d e f g = IIn7 (iF (IFix7 iF) a b c d e f g)
```

4.2.2 Indexed Data Types à la Carte

To build the AST, the data types à la carte [25] approach is taken. This allows for a modular approach, making the library more extendable later on. To be able to use this approach, it needs to be modified to support the 7 type indicies — a through to g:

```
data (iF :+: iG) (f' :: i → j → k → l → m → n → o → Type)
                 (a :: i) (b :: j) (c :: k) (d :: l) (e :: m) (f :: n) (g :: o) where
  L :: iF  f' a b c d e f g → (iF :+: iG) f' a b c d e f g
  R :: iG f' a b c d e f g → (iF :+: iG) f' a b c d e f g
infixr :+:
```

As mentioned in Section 2.4, using the :+: operator comes with problem of many L's and R's, when trying to create the AST structure. To avoid this, the :≺: operator can also be extended to work with $\mathsf{IFunctor}_7$. The definition follows closely to the original definition, with some minor modifications to the types and constraints.

```
class (IFunctor7 iF, IFunctor7 iG) ⇒ iF :≺: iG where
  inj :: iF f' a b c d e f g → iG f' a b c d e f g
instance IFunctor7 iF ⇒ iF :≺: iF where
  inj = id
instance (IFunctor7 iF, IFunctor7 iG) ⇒ iF :≺: (iF :+: iG) where
  inj = L
instance (IFunctor7 iF, IFunctor7 iG, IFunctor7 iH, iF :≺: iG) ⇒ iF :≺: (iH :+: iG) where
  inj = R · inj
```

Defining a constructor Data types for each constructor can 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 ::  [Type → Type] → [Type] → [Type]
               → [Type → Type] → [Type] → [Type] → Nat → Type)
          (inputsS  :: [Type → Type]) (inputsT  :: [Type]) (inputsA  :: [Type])
          (outputsS :: [Type → Type]) (outputsT :: [Type]) (outputsA :: [Type])
          (ninputs :: Nat) where
  Then :: (DataStore' fs as, DataStore' gs bs, DataStore' hs cs)
    ⇒ iF fs  as (Apply fs  as) gs bs (Apply gs bs) nfs
    → iF gs  bs (Apply gs bs) hs cs (Apply hs cs) ngs
    → Then iF fs as (Apply fs as) hs cs (Apply hs cs) nfs
```

Each iF denotes the recursive points in the data type, with the subsequent type arguments mirroring those seen in Section 3.4.1. A corresponding $\mathsf{IFunctor}_7$ instance formalises the points of recursion, by showing how to transform the structure inside it.

```
instance IFunctor7 Then where
  imap7 f (Then x y) = Then (f x) (f y)
```

The smart constructor, that injects the L's and R's automatically can be defined for Then as:

```
(<=>) :: (Then :≺: iF, DataStore' fs as, DataStore' gs bs, DataStore' hs cs)
   ⇒ IFix7 iF fs  as (Apply fs  as) gs bs (Apply gs bs) nfs
   → IFix7 iF gs bs (Apply gs bs) hs cs (Apply hs cs) nhs
   → IFix7 iF fs  as (Apply fs  as) hs cs (Apply hs cs) nfs
(<=>) l r = IIn7 (inj (Then l r))
infixr 4 <=>
```

The constructor adds one extra constraint, to the constructor defined in Section 3.4.1 — Then :≺: iF. This allows the smart constructor to produce an 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 then they can be combined together to form the CircuitF type, which can be used to represent a circuit.

```
type CircuitF = Id :+: Replicate :+: Then :+: Beside :+: Swap :+: DropL :+: DropR :+: Task :+: Map
```

The fixed-point of the CircuitF datatype can be defined with IFix_7:

```
type Circuit = IFix7 CircuitF
```

4.3 Process Network

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. The standard implementation of a process network will use 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.

4.3.1 Network Typeclass

To allow for different process networks, a typeclass will be used to specify all the functions that every network should have. The Network typeclass is defined as:

```
class Network n where
  startNetwork :: Circuit inputsS inputsT inputsA outputsS outputsT outputsA nInputs
               → IO (n   inputsS inputsT inputsA outputsS outputsT outputsA)
  stopNetwork  :: n inputsS inputsT inputsA outputsS outputsT outputsA
               → IO ()
  write :: HList' inputsS inputsT
        → n inputsS inputsT inputsA outputsS outputsT outputsA
        → IO ()
  read  :: n inputsS inputsT inputsA outputsS outputsT outputsA
        → IO (HList' outputsS outputsT)
```

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 4.4.
- stopNetwork is for cleaning up the network after it is no longer needed. For example, this could be stopping the 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.

Examples of how to use a network are included in Chapter 5.

4.3.2 The Basic Network Representation

An implementation of the Network typeclass is a BasicNetwork. This implementation makes use of a special case of heterogeneous list. A PipeList is used to represent a heterogeneous list of channels. This allows the BasicNetwork to store multiple channels in the same list with out the need for existential types — one of the problems previously encountered with chains in Section 3.3.

```
data PipeList (fs :: [Type → Type]) (as :: [Type]) (xs :: [Type]) where
  PipeCons :: Chan (f a) → PipeList fs as xs → PipeList (f ′ : fs) (a ′ : as) (f a ′ : xs)
  PipeNil  :: PipeList ′[] ′[] ′[]
```

Making use of PipeLists, the BasicNetwork data type can be defined. This definition makes use of record syntax, this allows for named fields, with accessors automatically generated.

```
data BasicNetwork (inputsS  :: [Type → Type]) (inputsT  :: [Type]) (inputsA  :: [Type])
                  (outputsS :: [Type → Type]) (outputsT :: [Type]) (outputsA :: [Type]) where
  BasicNetwork :: {
    threads :: [ThreadId],
    inputs  :: PipeList inputsS  inputsT  inputsA,
    outputs :: PipeList outputsS outputsT outputsA }
    → BasicNetwork inputsS inputsT inputsA outputsS outputsT outputsA
```

The BasicNetwork has three fields:

- threads is a list of ThreadIds, this allows for the threads to be managed after their creation.
- inputs is a PipeList containing the channels that the initial input into the network.
- outputs is a PipeList that stores channels, which the output of the network can be read from.

The Network type instance for a BasicNetwork is relatively trivial to implement: if given a function to transform a Circuit to it.

```
instance Network BasicNetwork where
  startNetwork   = buildBasicNetwork   -- Definition to come...
  stopNetwork n  = forM_ (threads n) killThread
  write uuid xs n = writePipes xs (inputs n)
  read n         = readPipes (outputs 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.

4.4 Translation to a Network

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 IO actions. The current definition for the fold icata_7 is not able perform monadic computation inside the algebra. To solve this unsafePerformIO could be used, however, for this to be safe the IO computation needs to have no side-effects. This fold will violate this rule, therefore, the only other way to support this is to modify the catamorphism to support monadic computation.

4.4.1 Indexed Monadic Catamorphism

An indexed monadic catamorphism, found in Section 2.3.1, can be used to perform this fold: it will allow for monadic computation within the algebra. However, icataM needs to be modified to support the 7 type indicies needed. The first step is to define a monadic imap that supports the needed number of type indicies. This will be added by extending the IFunctor_7 instance to also include a function named imapM_7.

$$\begin{array}{l}
\textbf{class}\ \mathsf{IFunctor_7\ iF}\ \textbf{where} \\
\quad \mathsf{imap_7} :: (\forall \mathsf{a\ b\ c\ d\ e\ f\ g}.\mathsf{f'\ a\ b\ c\ d\ e\ f\ g} \to \mathsf{g'\ a\ b\ c\ d\ e\ f\ g}) \to \mathsf{iF\ f'\ a\ b\ c\ d\ e\ f\ g} \to \mathsf{iF\ g'\ a\ b\ c\ d\ e\ f\ g} \\
\quad \mathsf{imapM_7} :: \mathsf{Monad\ m} \Rightarrow (\forall \mathsf{a\ b\ c\ d\ e\ f\ g}.\mathsf{f'\ a\ b\ c\ d\ e\ f\ g} \to \mathsf{m\ (g'\ a\ b\ c\ d\ e\ f\ g)}) \\
\qquad\qquad\qquad\qquad\qquad \to \mathsf{iF\ f'\ a\ b\ c\ d\ e\ f\ g} \\
\qquad\qquad\qquad\qquad\qquad \to \mathsf{m\ (iF\ g'\ a\ b\ c\ d\ e\ f\ g)}
\end{array}$$

The definition for this new function $\mathsf{imapM_7}$ for each instance closely follows the non-monadic version, however, now has a monadic function to map on the input. Here is the definition of the new $\mathsf{IFunctor_7}$ instance for Beside:

$$\begin{array}{l}
\textbf{instance}\ \mathsf{IFunctor_7\ Beside}\ \textbf{where} \\
\quad \mathsf{imap_7}\quad \mathsf{f\ (Beside\ l\ r)} = \mathsf{Beside\ (f\ l)\ (f\ r)} \\
\quad \mathsf{imapM_7\ f\ (Beside\ l\ r)} = \textbf{do} \\
\qquad \mathsf{l'} \leftarrow \mathsf{f\ l} \\
\qquad \mathsf{r'} \leftarrow \mathsf{f\ r} \\
\qquad \textbf{return}\ \mathsf{(Beside\ l'\ r')}
\end{array}$$

The definition is intuitively the same, just using do-notation instead.

Now that there is a indexed monadic map, it is possible to define the a monadic catamorphism for an $\mathsf{IFunctor_7}$:

$$\begin{array}{l}
\mathsf{icataM_7} :: (\mathsf{IFunctor_7\ iF}, \mathsf{Monad\ m}) \\
\quad \Rightarrow (\forall \mathsf{a\ b\ c\ d\ e\ f\ g}.\mathsf{iF\ f'\ a\ b\ c\ d\ e\ f\ g} \to \mathsf{m\ (f'\ a\ b\ c\ d\ e\ f\ g)}) \\
\quad \to \mathsf{IFix_7\ iF\ a\ b\ c\ d\ e\ f\ g} \\
\quad \to \mathsf{m\ (f'\ a\ b\ c\ d\ e\ f\ g)} \\
\mathsf{icataM_7\ algM\ (IIn_7\ x)} = \mathsf{algM} \mathrel{=\!\!\ll} \mathsf{imapM_7\ (icataM_7\ algM)\ x}
\end{array}$$

$\mathsf{icataM_7}$ is almost identical to icataM, however, it makes use of $\mathsf{imapM_7}$ instead of imapM.

4.4.2 BuildNetworkAlg

To use $\mathsf{icataM_7}$ to fold a Circuit into a BasicNetwork, an algebra is required. However, a standard algebra will not be able to complete this transformation. Consider this example Circuit with two tasks executed in sequence.

$$\begin{array}{l}
\mathsf{example} = \mathsf{task1} \\
\qquad\qquad\quad <\!\!> \\
\qquad\qquad\quad \mathsf{task2}
\end{array}$$

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.

$$\begin{array}{l}
\mathsf{foldl} :: (\mathsf{b} \to \mathsf{a} \to \mathsf{b}) \to \mathsf{b} \to [\mathsf{a}] \to \mathsf{b} \\
\mathsf{foldl\ f\ b\ as} = \mathsf{foldr}\ (\lambda \mathsf{a\ g\ x} \to \mathsf{g\ (f\ x\ a)})\ \mathsf{id\ as\ b}
\end{array}$$

A simple example of foldl can be considered.

$$\begin{array}{l}
\quad \mathsf{foldl}\ (+)\ 0\ [1,2] \\
\equiv \\
\quad (\lambda \mathsf{x} \to (\lambda \mathsf{x} \to \mathsf{id}\ (\mathsf{x}+2))\ (\mathsf{x}+1))\ 0
\end{array}$$

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 f g.

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

This newtype has 3 additional type parameters at the beginning, namely: asS, asT, asA. 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 An algebra type class can now be defined. This will ensure that the approach remains with modular: a new instance can be made when adding a new constructor to the language.

```
class (Network n, IFunctor7 iF) ⇒ BuildNetworkAlg n iF where
  buildNetworkAlg :: iF (  AccuN n asS asT asA) bsS bsT bsA csS csT csA nbs
                  → IO ((AccuN n asS asT asA) bsS bsT bsA csS csT csA nbs)
```

This algebra type class is parameterised by 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_7 that this instance is being defined for, an example is Then or Id. This algebra uses the N data type to perform an accumulating fold. The input to the algebra is an IFunctor_7 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.

To be able to define the algebra on sums of IFunctor_7s, without having a nest of Ls and Rs to pattern match, an instance for the sum of two data types is defined:

```
instance (BuildNetworkAlg n iF, BuildNetworkAlg n iG) ⇒ BuildNetworkAlg n (iF :+: iG) where
  buildNetworkAlg (L x)  = buildNetworkAlg x
  buildNetworkAlg (R y)  = buildNetworkAlg y
```

This instance enforces that there must also be an instance for the left and right hand side of the sum. It will then be able to automatically recurse through the Ls and Rs, to get to the types that have been summed.

Sketchy :s

The Initial Network Before being able to define the actual translation, there is one more base to cover. This is an accumulating fold that depends on the previous layer to be able to define the current one. However, what happens on the first layer? There is no previous Network to use. The initial network should have matching input and output types, this means that the input channels should be the same as the output channels.

To generate the PipeList of channels that will be stored in the initial network a new type class is defined:

```
class InitialPipes (inputsS :: [Type → Type]) (inputsT :: [Type]) (inputsA :: [Type]) where
  initialPipes :: IO (PipeList inputsS inputsT inputsA)
```

This type class is able to construct an initialPipes based on the type required, in the initial network. To be able to construct this value two instances are defined:

```
instance InitialPipes '[] '[] '[] where
  initialPipes = return PipeNil
instance InitialPipes fs as xs ⇒ InitialPipes (f ': fs) (a ': as) (f a ': xs) where
  initialPipes = do
    c ← newChan :: IO (Chan (f a))
    PipeCons c <$> (initialPipes :: IO (PipeList fs as xs))
```

The first instance deals with the base case: when the type lists are empty, an empty PipeList is created. The later more interesting case deals with a cons in the type lists. Here a new channel is created with the same type as the type removed from the front of the type list. The channel is then consed to the front of a PipeList with the remaining list generated by a recursive call.

Now that there is a method for creating a PipeList that matches a type-list, the initial network is defined:

```
initialNetwork
  :: ∀inputsS inputsT inputsA
    .(InitialPipes inputsS inputsT inputsA)
  ⇒ IO (BasicNetwork inputsS inputsT inputsA inputsS inputsT inputsA)
initialNetwork = do
  ps ← initialPipes :: IO (PipeList inputsS inputsT inputsA)
  return $ BasicNetwork [] ps ps
```

This creates a network that has matching input and output types. To do so a PipeList of the initial channels is created, which is then used as both the inputs and outputs of the network.

4.4.3 The Translation

Now that the algebra type class, and the initial input to the accumulating fold is defined, each instance of the type class are 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 → d
      output ← swapOutput (outputs n
      return $ BasicNetwork (threads n) (inputs n) output
    )
    where
      swapOutput :: PipeList '[f, g] '[a, b] '[f a, g b]
          → IO (PipeList '[g, f] '[b, a] '[g b, f a])
      swapOutput (PipeCons c1 (PipeCons c2 PipeNil)) =
        return $ 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 is able to transform the outputs and build a new BasicNetwork. To transform the networks it makes use of a function named swapOutput, which unpacks the PipeList and swaps the two channels c_1 and c_2 over.

Another basic constructor is Replicate: the purpose of this constructor is to duplicate the input to produce two outputs. The instance for Replicate is defined as:

```
instance BuildNetworkAlg BasicNetwork Replicate where
  buildNetworkAlg Replicate = return $ AccuN
    (λn → do
      output ← dupOutput (outputs n)
      return $ BasicNetwork (threads n) (inputs n) output
    )
    where
      dupOutput :: PipeList '[f] '[a] '[f a]
          → IO (PipeList '[f, f] '[a, a] '[f a, f a])
      dupOutput (PipeCons c PipeNil) = do
        c' ← dupChan c
        return $ PipeCons c (PipeCons c' PipeNil)
```

This instance follows a similar pattern to the Swap instance — defining a function which retrieves the accumulated network, then manipulates the outputs. However, the dupOutput function also has to make use of an operation on the channel — dupChan. This will create a new channel that will mirror the inputs of the original channel.

All other basic constructors will follow this pattern:

- Id, will return the same outputs as the accumulated network.
- DropL, will drop the *first* item in the output PipeList of the accumulated network.
- DropR, will drop the *last* item in the output PipeList of the accumulated network.

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:

```
taskExecutor
  :: Task iF inputsS inputsT inputsA outputS outputT outputsA ninputs
  → PipeList inputsS inputsT inputsA
  → PipeList outputS outputT outputA
  → IO ()
taskExecutor (Task f outStore) inPipes outPipes = forever
  (do
    taskInput ← readPipes inPipes
    r ← f taskInputs outStore
    writePipes (HCons' r HNil') outPipes
  )
```

The taskExecutor has three arguements:

- The Task to be executed on the thread.
- A PipeList which has channels containing the input values.
- A PipeList to output the results of the Task.

A taskExecutor 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, using the aptly named function forever.

Making use of the taskExecutor, the algebra instance for Task is defined as:

```
instance BuildNetworkAlg BasicNetwork Task where
  buildNetworkAlg (Task t out) = return $ AccuN
    (λn → do
      c ← newChan
      let output = PipeCons c PipeNil
      threadId ← forkIO (taskExecutor (Task t out) (outputs n) output)
      return $ BasicNetwork (threadId : threads 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 $ N
    (λn → do
      nx ← fx n
      fy nx
    )
```

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 nx, 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 difficult algebra to define as the accumulated network needs to be split in half to pass to the two recursive sides of Beside. An instance of the algebra is defined as:

```
instance BuildNetworkAlg BasicNetwork Beside where
  buildNetworkAlg = beside
```

This requires a beside function, however to define this function some extra tools are required. The first is takeP, which will take the first n elements from a PipeList:

```
takeP :: SNat n → PipeList fs as xs → PipeList (Take n fs) (Take n as) (Take n xs)
takeP SZero     _                = PipeNil
takeP (SSucc _) PipeNil          = PipeNil
takeP (SSucc n) (PipeCons x xs)  = PipeCons x (takeP n xs)
```

This makes use of the Take type family to take n elements from each of the type lists: fs, as, and xs. It follows the same structure as the take :: Int → [a] → [a] defined in the Prelude.

The next function is dropP, it drops n elements from a PipeList:

```
dropP :: SNat n → PipeList fs as xs → PipeList (Drop n fs) (Drop n as) (Drop n xs)
dropP SZero     l                = l
dropP (SSucc _) PipeNil          = PipeNil
dropP (SSucc n) (PipeCons _ xs)  = dropP n xs
```

This function again follows the same structure as drop :: Int → [a] → [a] defined in the Prelude. Both takeP and dropP are used to split the outputs of a network after n elements. This requires the knowledge of what n is at the value level, however n is only stored at the type level as the argument ninputs. To be able to recover this value the IsNat type class, as defined in Section 2.5.2 is used. The recoverNInputs function is able to direct the IsNat type class to the correct type argument, and produces an SNat with the same value as that stored in the type.

```
recoverNInputs :: (Length bsS∼Length bsT, Length bsT∼Length bsA, Length bsA∼Length bsS,
                   ninputs∼Length bsS, IsNat ninputs, Network n)
  ⇒ (N n asS asT asA) bsS bsT bsA csS csT csA (ninputs :: Nat)
  → SNat (Length bsS)
circuitInputs _ = nat
```

After splitting a network and generating two new networks, the outputs will need to be joined together again: this will require the appending of two PipeLists. To do this an AppendP type class is defined:

```
class AppendP fs as xs gs bs ys where
  appendP :: PipeList fs as xs → PipeList gs bs ys → PipeList (fs :++ gs) (as :++ bs) (xs :++ ys)
```

This type class has one function appendP, it is able to append two PipeLists together. It makes use of the :++ type family to append the type lists together. The instances for this type class are made up of two cases: the base case and a recursive case.

```
instance AppendP '[] '[] '[] gs bs ys where
  appendP PipeNil         ys = ys
instance (AppendP fs as xs gs bs ys) ⇒ AppendP (f ': fs) (a ': as) (f a ': xs) gs bs ys where
  appendP (PipeCons x xs) ys = PipeCons x (appendP xs ys)
```

The base case corresponds to having an empty list on the left, with some other list on the right. Here the list on the right is returned. The recursive case, simply takes 1 element from the left hand side and conses it onto the from of a recursive call, with the rest of the left hand side.

It is now possible to define the beside function. The result is calculated in 4 steps, with helper functions for each step:

1. Get the number of inputs (ninputs) on the left hand side of the Beside constructor. This will give the information needed to split the inputted accumulated network n.

2. Split the network into a left and right hand side. This will retain the same input type to the network, as there is no information on how to split that. Only the output PipeList will be split into two parts.

3. Translate the both the left and right network. This will perform the recursive step and generate two new networks with the networks from the left and right added to the accumulated network n.

4. Join the networks back together. Now that the left and right hand side of this layer has been added to the accumulated network, the two sides need to be joined back together to get a single network that can be returned.

This is very long, but I cant really break it up because it needed the scoped type variables from beside

```
beside :: ∀asS asT asA bsS bsT bsA csS csT csA (nbs :: Nat)
  .Beside (AccuN BasicNetwork asS asT asA) bsS bsT bsA csS csT csA nbs
   → IO ((AccuN BasicNetwork asS asT asA) bsS bsT bsA csS csT csA nbs)
beside (Beside l r) = return $ AccuN
  (λn → do
    let ninputs = circuitInputs l
    (nL, nR) ← splitNetwork ninputs n
    (newL, newR) ← translate ninputs (nL, nR) (l, r)
    joinNetwork (newL, newR)
  )
where
  splitNetwork :: SNat nbsL
     → BasicNetwork asS asT asA bsS bsT bsA
     → IO (BasicNetwork asS asT asA (Take nbsL bsS) (Take nbsL bsT) (Take nbsL bsA),
           BasicNetwork asS asT asA (Drop nbsL bsS) (Drop nbsL bsT) (Drop nbsL bsA))
  splitNetwork nbs n = return
    (BasicNetwork (threads n) (inputs n) (takeP nbs (outputs n)),
     BasicNetwork (threads n) (inputs n) (dropP nbs (outputs n)))
  translate :: SNat nbsL
     → (BasicNetwork asS asT asA (Take nbsL bsS) (Take nbsL bsT) (Take nbsL bsA),
        BasicNetwork asS asT asA (Drop nbsL bsS) (Drop nbsL bsT) (Drop nbsL bsA))
     → ((AccuN BasicNetwork asS asT asA)
          (Take nbsL bsS) (Take nbsL bsT) (Take nbsL bsA) csLS csLT csLA nbsL,
        (AccuN BasicNetwork asS asT asA)
          (Drop nbsL bsS) (Drop nbsL bsT) (Drop nbsL bsA) csRS csRT csRA nbsR)
     → IO (BasicNetwork asS asT asA csLS csLT csLA,
           BasicNetwork asS asT asA csRS csRT csRA)
  translate _ (nL, nR) (N cL, N cR) = do
    nL′ ← cL nL
    nR′ ← cR nR
    return (nL′, nR′)
  joinNetwork :: (AppendP csLS csLT csLA csRS csRT csRA)
     ⇒ (BasicNetwork asS asT asA csLS csLT csLA, BasicNetwork asS asT asA csRS csRT csRA)
     → IO (BasicNetwork asS asT asA (csLS :++ csRS) (csLT :++ csRT) (csLA :++ csRA))
  joinNetwork (nL, nR) = return
     $ BasicNetwork (nub (threads nL ++ threads nR)) (inputs nL) (outputs nL `appendP` outputs nR)
```

The splitNetwork function creates two new BasicNetworks. To split the output values, takeP, and dropP are used. translate performs the recursive step in the accumulating fold, which produces two new networks that include this layer. joinNetwork takes the two new networks and appends the outputs with appendP. It also has to append the thread ids from both sides, however, this will now include duplicates as threads were not split in splitNetwork. To combat this nub is used, which returns a list containing all the unique values in the original.

Define buildBasicNetwork

4.5 UUIDS

When inputting multiple values into a Network problems can occur. For example, a task's output pointer is statically defined. If this were a file, it would result in files being overwritten before they have been read. This is eliminated through the use of UUIDs, they act as a unique identifier for each input into the network.

4.5.1 Modifications

To be able to support a UUID, several small modifications need to be made.

Data Store The value is accessible to the data store when saving by making a small modification:

```
class DataStore f a where
  fetch :: UUID → f a → IO a
  save :: UUID → f a → a → IO (f a)
```

This can then be made use of when reading or writing to a data store. For example, a filename could be prepended with the unique identifier, or it could be used as a primary key when saving to a database table.

PipeList To transfer the value around the network, the PipeList data type is modified to store channels of type (UUID, f a), instead of just f a:

```
data PipeList (fs :: [Type → Type]) (as :: [Type]) (xs :: [Type]) where
  PipeCons :: Chan (UUID, f a) → PipeList fs as xs → PipeList (f ': fs) (a ': as) (f a ': xs)
  PipeNil :: PipeList '[] '[] (Apply '[] '[])
```

Task Executor The task executor needs to be modified, so that it retries the UUID, gives it to the task and passes it on with the output.

```
taskExecutor :: Task iF inputsS inputsT inputsA outputS outputT outputsA ninputs
   → PipeList inputsS inputsT inputsA
   → PipeList outputS outputT outputA
   → IO ()
taskExecutor (Task f outStore) inPipes outPipes = forever
  (do
    (uuid, taskInput) ← readPipes inPipes
    r                 ← f uuid taskInputs outStore
    writePipes uuid (HCons' r HNil') outPipes
  )
```

Network: Read & Write The read and write methods defined in the Network type class are modified to also take a UUID:

```
class Network n where
   ...
  read :: n inputsS inputsT inputsA outputsS outputsT outputsA → IO (UUID, HList' outputsS outputsT)
  write :: UUID → HList' inputsS inputsT → n inputsS inputsT inputsA outputsS outputsT outputsA → IO ()
```

4.5.2 Helper Functions

There are several helper functions for reading and writing into a network:

```
input :: Network n ⇒ HList' inputsS inputsT
  → n inputsS inputsT inputsA outputsS outputsT outputsA
  → IO UUID
input_ :: Network n ⇒ HList' inputsS inputsT
  → n inputsS inputsT inputsA outputsS outputsT outputsA
  → IO ()
output_ :: Network n ⇒ n inputsS inputsT inputsA outputsS outputsT outputsA
  → IO (HList' outputsS outputsT)
```

Both of the input functions generate a random UUID, meaning the user does not have to specify one. input will return this generated values, whereas input_ will not. output_ fetches values from a network, but does not return the UUID with the outputs.

4.6 Failure in the Process Network

Currently, when an exception occurs in a task the whole network crashes — this isn't desired behaviour. There are many ways to model failure in Haskell, one such example could be the Maybe monad.

4.6.1 Maybe not Maybe

Maybe can capture failure, with Just x being the success case, and Nothing being the failed case. When Nothing is produced the rest of the computation automatically fails due to the definition of ≫=.

```
instance Monad Maybe where
  return x = Just x
  (≫=) Nothing _ = Nothing
  (≫=) (Just x) f = f x
```

This would work well in a network as an error in one task can be propagated to all it dependents, causing them to fail gracefully, and the error propagating further through the network. Howver, there is one problem with Maybe, it does not retain any information about the error that occured. This information would be very useful to a user, as it helps them debug the issue.

4.6.2 Except Monad

The Except monad is based on Either, with the Left constructor representing failure, and the Right constructor indicating success. This means that an error message can now be stored, when a failure ultimately occurs. Since tasks already execute in the IO monad, a monad transformer ExceptT is required, so that failure can be implemented into the network. This allows for computation in both IO, and Except, however, any IO computation will need to be lifted into the ExceptT monad.

The following modifications are required to add modelling of failure in a network.

PipeList A pipelist will now need to also transfer information about whether the previous task failed to execute. To do this it will carry an Either TaskError (f a), with TaskError being a custom data type storing the error message text.

```
data PipeList (fs :: [Type → Type]) (as :: [Type]) (xs :: [Type]) where
  PipeCons :: Chan (UUID, Either TaskError (f a))
    → PipeList fs as xs
    → PipeList (f ' : fs) (a ' : as) (f a ' : xs)
  PipeNil :: PipeList '[] '[] (Apply '[] '[])
```

Task Executor The task executor will need to be modified so that it executes the tasks in the ExceptT monad.

```
taskExecuter :: Task iF inputsS inputsT inputsA outputS outputT outputsA ninputs
    → PipeList inputsS inputsT inputsA
    → PipeList outputsS outputT outputA
    → IO ()
taskExecuter (Task f outStore) inPipes outPipes = forever
  (do
    (uuid, taskInputs) ← readPipes inPipes
    r                  ← runExceptT $
      (do
        input ← (ExceptT · return) taskInputs
        r     ← catchE (intercept (f uuid input outStore))
                  (throwE · TaskError · ExceptionMessage · displayException)
        return (HCons' (r `deepseq` r) HNil')
      )
    writePipes uuid r outPipes
  )
```

This version of the executor, first reads the values from the input channels. It then runs some computation in the ExceptT monad to get a return value r :: Either TaskError (HList' outputsS outputsT). The return value is then sent along the output channels.

The catchE function can be used to catch an exception thrown in a task, the exception is then converted to a TaskError, and re-thrown. This will mean that only a Either TaskError (HList' outputsS outputsT) is returned, rather than another type of error.

There is, however, an error that is caused by Haskell's laziness: if a value is not evaluated inside the runExceptT block then it will not be caught, and the program will continue to crash. To solve this the deepseq function is used, which fully evaluates the left argument, before returning the right argument. This forces any error that could occur to happen inside the runExceptT block.

4.7 Evaluation

- **Type-safe** — This is a continuation of the previous requirement for the language. It is also important that once the user has built a well-typed Circuit, that the code also continues to be executed in a well-typed environment, to ensure that all inputs and outputs are correctly typed.
- **Parallel** — One of the key benefits that comes from dataflow programming is implicit parallelisation. With this DSL being tailored towards data pipelines, which could be computationally expensive, it should be able to benefit from parallel execution.
- **Competitive Speed** — This library should be able to execute dataflows in a competitive time, with other libraries that already exist.
- **Failure Tolerance** — It is important that if one invocation of a task crashes, it does not crash the whole program. This implementation should be able to gracefully handle errors and propagate them through the circuit.
- **Usable** — The implementation of the library should not break any of the usability of the language design.
- **Maintainable** — It should be easy to maintain the library and add new constructors in the future.

☑ **Type-safe**

☑ **Parallel**

☑ **Competitive Speed** The network is, indeed, able to compute results in a time that is competitive with other libraries. More detail on this can be found in Chapter 6.

☑ **Failure Tolerance**

☑ **Usable**

Chapter 5

Examples

5.1 Machine Learning (Audio Playlist Generation)

A use case for CircuitFlow is when building data pipelines. Here there are many tasks that can only be executed when other data files have been produced. A data pipeline will also benefit from being parallel to improve run-times. CircuitFlow can help to build a parallel data pipeline, without the user having to worry about how to combine all their data manipulations together, in a way that will not run into concurrency problems.

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 your songs based on the top 10 artists and songs that you have listened to over the last 3 months. However, each of the months data is stored in different files that need aggregating together, before they can be input into the model.

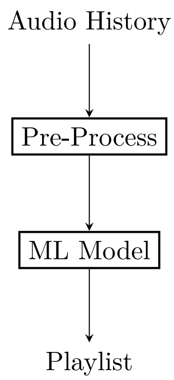


Figure 5.1: A dataflow diagram for playlist generation

5.1.1 Building the pre-processing Circuit

This circuit will need to have 3 different inputs — each months listening history. It will also have 2 outputs: the top 10 songs and artists. To begin with a dataflow diagram can be constructed — seen in Figure 5.2. This will model all the dependencies between each of the pre-processing tasks.

To write this in CircuitFlow, the first step is to create all the tasks that will be used. Both the AggSongs and AggArtists tasks will require three inputs, which will be CSVs. This means that it is possible to make use of the pre-defined data store: NamedCSVStore. The type instances that are required to parse the CSV will be omitted, as they are not relevant to the discussion.

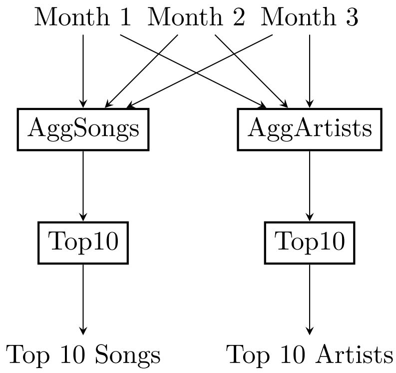


Figure 5.2: A dataflow diagram for pre-processing the song data

```
aggSongsTask :: Circuit
  '[NamedCSVStore,          NamedCSVStore,          NamedCSVStore]
  '[                 [Listen],                [Listen],                [Listen]]
  '[NamedCSVStore [Listen], NamedCSVStore [Listen], NamedCSVStore [Listen]]
  '[VariableStore]
  '[              [TrackCount]]
  '[VariableStore [TrackCount]]
  N3
aggSongsTask = multiInputTask f Empty
where
  f :: HList ('[[Listen], [Listen], [Listen]] → [TrackCount]
  f (HCons month1 (HCons month2 (HCons month3 HNil))) =
    (map (uncurry TrackCount) · reverse · count_ · map track) (month1 ++ month2 ++ month3)
```

This task applies a composition of functions:

1. Firstly, the track is extracted from the Listen data type using the function track :: Listen → Track.
2. Next, a list of unique tracks are extracted, along with the number of appearaces they made in the original list. This makes use of a special variant count_ :: [a] → [(a, Int)], which will also sort the list in ascending order.
3. This list is then flipped to descending order with reverse :: [a] → [a].
4. Finally, a TrackCount value is made from the previous tuple uncurry TrackCount :: (Track, Int) → TrackCount.

```
aggArtistsTask :: Circuit
  '[NamedCSVStore,          NamedCSVStore,          NamedCSVStore]
  '[                 [Listen],                [Listen],                [Listen]]
  '[NamedCSVStore [Listen], NamedCSVStore [Listen], NamedCSVStore [Listen]]
  '[VariableStore]
  '[              [ArtistCount]]
  '[VariableStore [ArtistCount]]
  N3
aggArtistsTask = multiInputTask f Empty
where
  f :: HList '[[Listen], [Listen], [Listen]] → [ArtistCount]
  f (HCons month1 (HCons month2 (HCons month3 HNil))) =
    (map (uncurry ArtistCount) · reverse · count_ · map (artist · track)) (day1 ++ day2 ++ day3)
```

The aggArtistsTask is constructed in a similar way to the aggSongsTask, however, it extracts the artist from the track before aggregating the data.

The final task to define is Top10, however, this task is used multiple times. Therefore, it would be beneficial if the type was polymorphic so that it can receive and input of both [ArtistCount] and [TrackCount]. This is possible to do:

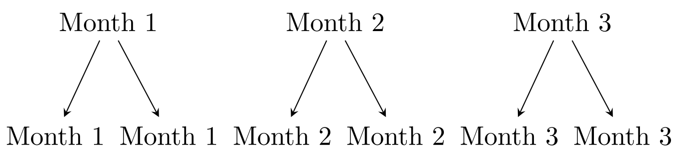
```
top10Task :: (ToNamedRecord a, FromNamedRecord a, DefaultOrdered a)
  ⇒ FilePath
  → Circuit '[VariableStore]    '[[a]] '[VariableStore    [a]]
            '[NamedCSVStore] '[[a]] '[NamedCSVStore [a]]
            N1
top10Task filename = functionTask (take 10) (NamedCSVStore filename)
```

The top10Task takes an input list and then returns the first 10 from the list.

Now that all of the tasks have been defined, they need to be combined into a circuit. The dataflow diagram seen in Figure 5.2, will prove to be helpful. The first obvious problem is how to create a circuit that can achieve the top part of the diagram, transforming the inputs so that they can be passed into two tasks. This can be dealt with in layers: using the diagrams associated with each constructor, it is possible to convert a layer into a composition of Circuits. The first layer, seen in Figure 5.3 will be responsible for duplicating each of the three inputs.

$$\mathsf{layer1} = \mathsf{replicate} <> \mathsf{replicate} <> \mathsf{replicate}$$

(a)



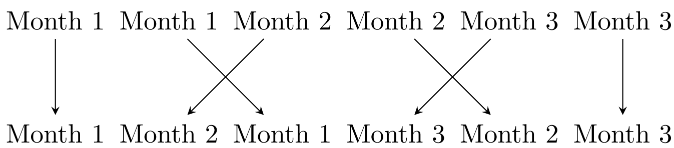
(b)

Figure 5.3: The first layer (a) and its corresponding dataflow diagram (b).

Scanning down the dataflow diagram the next layer to deal with is the two swaps that occur: month 1 and 2, and month 2 and 3. This can been seen in Figure 5.4

$$\mathsf{layer2} = \mathsf{id} <> \mathsf{swap} <> \mathsf{swap} <> \mathsf{id}$$

(a)



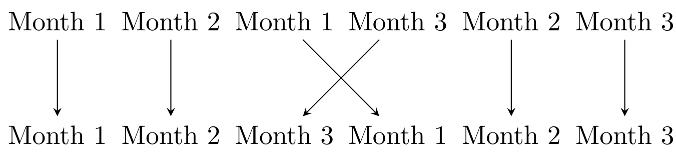
(b)

Figure 5.4: The second layer (a) and its corresponding dataflow diagram (b).

The final layer, seen in Figure 5.5, will swap month 1 with month 3.

```
layer3 = id <> id <> swap <> id <> id
```

(a)



(b)

Figure 5.5: The third layer (a) and its corresponding dataflow diagram (b).

These layers can be stacked on top of each other to provide a full transformation:

```
organiseInputs = layer1
                 <=>
                 layer2
                 <=>
                 layer3
```

The tasks can now be combined with this transformation to provide the full circuit:

```
preProcPipeline :: Circuit
  '[NamedCSVStore,         NamedCSVStore,         NamedCSVStore]
  '[              [Listen],               [Listen],               [Listen]]
  '[NamedCSVStore [Listen], NamedCSVStore [Listen], NamedCSVStore [Listen]]
  '[NamedCSVStore, NamedCSVStore]
  '[              [ArtistCount],               [TrackCount]]
  '[NamedCSVStore [ArtistCount], NamedCSVStore [TrackCount]]
  N3
preProcPipeline = organiseInputs
                  <=>
                  aggSongsTask                  <> aggArtistsTask
                  <=>
                  top10Task "top10Artists.csv" <> top10Task "top10Artists.csv"
```

Again it can be seen how this structure of tasks directly correlates with the dataflow diagram previously seen in Figure 5.2. This helps to make it easier when designing circuits as it can be constructed visually level by level.

5.1.2 Building the prediction Circuit

Now that there is a pipeline to pre-process the data for the model, a new playlist can be created. Again it is possible to build a circuit that combines with the pre-processing circuit.

The audio company train models for each user and store them in a cloud storage service. This would be a good use-case for creating a new DataStore — called a ModelStore. Say that there are two types defined ModelStore and NewSongPlaylist: a new DataStore instance can be defined for each different model. Here is the NewSongsPlaylist as an example:

```
instance DataStore ModelStore NewSongsPlaylist where
    -- Fetch from cloud storage and load into the program
  fetch (ModelStore NewSongsPlaylist)   = ...
  save  (ModelStore NewSongsPlaylist) _ = ...
```

fetch is used to download a trained model from the cloud store. save will be used when training a new model, it will allow the model to be saved for use in the future.

With this DataStore a new task can be defined that will, take a users top 10 songs and artists, and a model as input. It will then output a list of songs to create a playlist.

```
predictTask :: Circuit
    '[NamedCSVStore,                NamedCSVStore, ,              ModelStore]
    '[              [ArtistCount],                [TrackCount],             NewSongsPlayList]
    '[NamedCSVStore [ArtistCount], NamedCSVStore [TrackCount], ModelStore NewSongsPlaylist]
    '[NamedCSVStore]
    '[              [Track]]
    '[NamedCSVStore [Track]]
    N3
predictTask = multiInputTask f (NamesCSVStore "newSongsPlaylist.csv")
  where
    f :: HList '[[ArtistCount], [TrackCount], NewSongsPlayList] → [Track]
    f (HCons topArtists (HCons topSongs (HCons model HNil))) =
      predict model topArtists topSongs
```

The predictTask can now be combined with the pre-processing circuit to create a circuit that is able to create a new playlist from listening history.

```
createPlaylist = preProcPipeline <> id
                  <>
                 predictTask
```

The createPlaylist, circuit can now be converted into a Network and used on user data to generate new playlists, based on their top songs. preProcPipeline will be revisited in Chapter **??**, to act as a benchmark to compare the performance of the CircuitFlow library.

5.2 Build System (lhs2TeX)

Another use case for a task based dependency system is a build system. For example, a Makefile is a way of specifying the target files from some source files, with a command that can be used to generate the target file. CircuitFlow: could also be used to model such a system.

Consider this dissertation, which is made using LaTeX. This project is made up of multiple sub-files, each written in a literate Haskell format. Each of these files needs to be pre-processed by the `lhs2TeX` command to produce the `.tex` source file. Once each of these files has been generated, then the LaTeXproject can be built into a PDF file.

5.2.1 Building the Circuit

The Circuit defined here makes use of the mapC operator. To do so a Circuit is defined that is able to build a single `.tex` file from a `.lhs`. This has to make use of the standard task constructor:

```
buildLhsTask :: Circuit '[VariableStore] '[String] '[VariableStore String]
                        '[VariableStore] '[String] '[VariableStore String]
                        N1
buildLhsTask = task f Empty
  where
    f :: UUID
      → HList' '[VariableStore] '[String]
      → VariableStore String
      → ExceptT SomeException IO (VariableStore String)
    f _ (HCons' (Var fInName) HNil') _ = do
      let fOutName = fInName −<.> "tex"
      lift (callCommand ("lhs2tex -o " ++ fOutName ++ " " ++ fInName))
      return (Var fOutName)
```

This Circuit makes use of the callCommand function from the System.Process library. This allows the task to execute external commands, that may not necessarily be defined in Haskell. A similar Circuit can be defined that will compile the `.tex` files and produce a PDF.

```
buildTexTask :: String → Circuit '[VariableStore] '[[String]] '[VariableStore [String]]
                                  '[VariableStore] '[String]   '[VariableStore String]
                                  N1
buildTexTask name = task f Empty
  where
    f :: UUID
      → HList' '[VariableStore] '[[String]]
      → VariableStore String
      → ExceptT SomeException IO (VariableStore String)
    f mainFileName (HCons' (Var _) HNil') _ = do
      lift
        (callCommand
          ("texfot -no-stderr latexmk -interaction=nonstopmode -pdf "
           ++ "-no-shell-escape -bibtex -jobname="
           ++ name
           ++ " "
           ++ mainFileName
          )
        )
      return (Var "dissertation.pdf")
```

The buildTexTask also demonstrates how it is possible to pass a global parameter into a task. This can allow tasks to be made in a more reusable way. Saving a user from defining multiple variations of the same task.
These two tasks can now be combined into a Circuit. This Circuit can be interpreted as inputting a list of `.lhs` files, which are sequentially compiled to `.tex` files by the mapC operator. These files are *then* built by the buildTexTask to produce a PDF file as output.

```
buildDiss :: String → Circuit '[VariableStore] '[[String]] '[VariableStore [String]]
                               '[VariableStore] '[String]   '[VariableStore String]
                               N1
buildDiss name = mapC buildLhsTask Empty
                   <->
                 buildTexTask name
```

mapC takes two arguments, the inner circuit to execute and a pointer to the location it should store its results. In this case the output data store is a VariableStore, therefore, the pointer to this location is just an Empty variable.

5.2.2 Using the Circuit

To use this Circuit, a Config data type is used to store the information needed within the system to build the project:

```
data Config = Config
  {mainFile    :: FilePath
  , outputName :: String
  , lhsFiles   :: [FilePath]
  }
  deriving (Generic, FromJSON, Show)
```

This data type uses record syntax to have name fields:

- mainFile is the name of the root file that should be used for compilation.
- outputName is the desired name for the output PDF file.
- lhsFiles are all the literate haskell files required to build the LaTeXdocument.

The Config data type also derives the Generic and FromJSON instance. This allows it to be used in conjunction with a YAML file to specify these parameters. The config can be loaded with:

```
loadConfig :: IO Config
loadConfig = loadYamlSettings ["dissertation.tex-build"] [] ignoreEnv
```

An example config file can be seen in Figure 5.6.

```
mainFile: dissertation.lhs
outputName: dissertation
lhsFiles: [dissertation.lhs
          , chapters/introduction.lhs
          , chapters/background.lhs
          , chapters/the-language.lhs
          , chapters/implementation.lhs
          , chapters/evaluation.lhs
          , chapters/examples.lhs]
```

Figure 5.6: An example config file for the lhs2TeX build system

To be able to use the system a main function is defined, which will serve as the entry point to the executable:

```
main :: IO ()
main = do
  config ← loadConfig
  n      ← startNetwork (buildDiss (outputName config)) :: IO
    (BasicNetwork '[VariableStore] '[[String]] '[VariableStore [String]]
                  '[VariableStore] '[String]   '[VariableStore String])
  write (mainFile config -<.> "tex") (HCons' (Var (lhsFiles config)) HNil') n
  _ ← read n
  stopNetwork n
```

The main function in the build system has 5 steps:

1. The config file is loaded.
2. A network is started based on the buildDiss circuit. The type of network to be started has to be annotated, so that the type system knows which Network instance to use.
3. The build job is input into the network, with the input values being a list of `.lhs` files to compile.
4. A call is made to the blocking function read, although the outputs are not needed, the call to read is. This prevents the program ending before the network has complete processing values.
5. Finally, the network is destroyed.

This dissertation makes use of this build system to be able include literate Haskell files.

5.3 Types saving the day

Consider an example shown in the docs for Luigi [23], that is made up of two tasks. The first generates a list of words and saves it to a file and second counts the number of letters in each of those words. The counting letters task is dependent on the words being generated.

Figure 5.7, shows an implementation of such a system, in the Python library called Luigi. However, this implementation has a very subtle bug! `GenerateWords` writes the words to a file separated by new lines, but `CountLetters` reads that same file as a comma-separated list. This shows a key flaw in this system, 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 run-time error, instead, it will just produce the incorrect result. For a developer this is extremely unhelpful, it means more of time is used writing tests — something that no one enjoys.

```
import luigi

class GenerateWords(luigi.Task):
    def output(self):
        return luigi.LocalTarget('words.txt')

    def run(self):
        # write a dummy list of words to output file
        words = ['apple', 'banana', 'grapefruit']

        with self.output().open('w') as f:
            for word in words:
                f.write('{word}\n'.format(word=word))

class CountLetters(luigi.Task):
    def requires(self):
        return GenerateWords()

    def output(self):
        return luigi.LocalTarget('letter_counts.txt')

    def run(self):
        # read in file as list
        with self.input().open('r') as infile:
            words = infile.read().split(',')

        # write each word to output file with its corresponding letter count
        with self.output().open('w') as outfile:
            for word in words:
                outfile.write('{word}:{letter_count}\n'.format(
                    word=word,
                    letter_count=len(word)
                ))
```

Figure 5.7: A Broken Luigi Example

The Fix Why not eliminate the need for all of this with DataStores and types. As previously mentioned in Section 3.2.1, a DataStore can be used to abstract the reading and writing of many different sources. This will help to ensure correctness of this step, by eliminating any possible duplicated code. Instead, just having the fetch and save methods to test.

The second greater benefit, is to use DataStores in combination with the type system. Each constructor for a Circuit will, enforce that the types of a DataStore align correctly. 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 as before can be seen in Figure 5.8. In this example it will fail to compile, giving the error:

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

This will benefit the user as it reduces the feedback loop of knowing if the program will succeed. Previously the whole data pipeline had to be run, whereas now this information can be informed to the user at compile-time.

```
generateWords :: Circuit '[VariableStore] '[()]        '[VariableStore ()]
                         '[FileStore]     '[[String]] '[FileStore     [String]]
                         N1
generateWords = functionTask (const ["apple", "banana", "grapefruit"]) (FileStore "fruit.txt")

countLetters :: Circuit '[CommaSepFile] '[[String]] '[CommaSepFile [String]]
                        '[FileStore]    '[[String]] '[FileStore    [String]]
                        N1
countLetters = functionTask (map f) (FileStore "count.txt")
  where
    f word = (concat [word, ":", show (length word)])

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

Figure 5.8: A Broken Circuit Example

Chapter 6

Benchmarks

6.1 Runtime comparison

6.1.1 Lazy evaluation problems

6.2 Use of Library

Something about how DataStores prevent the need for luigi.ExternalTask at the beginning.

6.2.1 Other Libraries

How does it compare?

Luigi Object-orientated approach Python library

Funflow Uses arrows to compose flows sequentially.

6.3 Type saftey

Chapter 7

Conclusion

Todo list

Bibliography

[1] Quartz composer user guide, Jul 2007.

[2] Patrick Bahr and Tom Hvitved. Compositional data types. In *Proceedings of the Seventh ACM SIGPLAN Workshop on Generic Programming*, WGP '11, page 83–94, New York, NY, USA, 2011. Association for Computing Machinery.

[3] Dr Peter Bentley. The end of moore's law: what happens next?, Apr 2020.

[4] James Cheney and Ralf Hinze. First-class phantom types. 01 2003.

[5] Bob Coecke, Tobias Fritz, and Robert W. Spekkens. A mathematical theory of resources. *Information and Computation*, 250:59–86, Oct 2016.

[6] Edward Costello. *Figure 15. Connecting a Math patch to the Sprite Width and Height inlets.* Creating Graphical User Interfaces for Csound using Quartz Composer. Csound Journal, Nov 2012.

[7] Richard A. Eisenberg, Dimitrios Vytiniotis, Simon Peyton Jones, and Stephanie Weirich. Closed type families with overlapping equations. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, page 671–683, New York, NY, USA, 2014. Association for Computing Machinery.

[8] Richard A. Eisenberg and Stephanie Weirich. Dependently typed programming with singletons. In *Proceedings of the 2012 Haskell Symposium*, Haskell '12, page 117–130, New York, NY, USA, 2012. Association for Computing Machinery.

[9] Felienne Hermans, Martin Pinzger, and Arie van Deursen. Breviz: Visualizing spreadsheets using dataflow diagrams, 2011.

[10] Ralf Hinze. An algebra of scans. In Dexter Kozen, editor, *Mathematics of Program Construction*, pages 186–210, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[11] Tiffany Hu. Building out the seatgeek data pipeline, Jan 2015.

[12] Patricia Johann and Neil Ghani. Foundations for structured programming with gadts. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '08, page 297–308, New York, NY, USA, 2008. Association for Computing Machinery.

[13] Gilles Kahn. The semantics of a simple language for parallel programming. In Jack L. Rosenfeld, editor, *Information Processing, Proceedings of the 6th IFIP Congress 1974, Stockholm, Sweden, August 5-10, 1974*, pages 471–475. North-Holland, 1974.

[14] Oleg Kiselyov, Ralf Lämmel, and Keean Schupke. Strongly typed heterogeneous collections. In *Proceedings of the 2004 ACM SIGPLAN Workshop on Haskell*, Haskell '04, page 96–107, New York, NY, USA, 2004. Association for Computing Machinery.

[15] S.M. Lane, S.J. Axler, Springer-Verlag (Nowy Jork)., F.W. Gehring, and P.R. Halmos. *Categories for the Working Mathematician.* Graduate Texts in Mathematics. Springer, 1998.

[16] E. A. Lee and T. M. Parks. Dataflow process networks. *Proceedings of the IEEE*, 83(5):773–801, 1995.

[17] Conor McBride. Functional pearl: Kleisli arrows of outrageous fortune. *Journal of Functional Programming (accepted for publication)*, 2011.

[18] John C. Mitchell and Gordon D. Plotkin. Abstract types have existential type. *ACM Trans. Program. Lang. Syst.*, 10(3):470–502, July 1988.

[19] G. E. Moore. Cramming more components onto integrated circuits, reprinted from electronics, volume 38, number 8, april 19, 1965, pp.114 ff. *IEEE Solid-State Circuits Society Newsletter*, 11(3):33–35, 2006.

[20] Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn. Simple unification-based type inference for gadts. *SIGPLAN Not.*, 41(9):50–61, September 2006.

[21] Tom Schrijvers, Simon Peyton Jones, Manuel Chakravarty, and Martin Sulzmann. Type checking with open type functions. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*, ICFP '08, page 51–62, New York, NY, USA, 2008. Association for Computing Machinery.

[22] Spotify. Luigi: https://github.com/spotify/luigi.

[23] Spotify. Tasks, Apr 2020.

[24] Josef Svenningsson and Emil Axelsson. Combining deep and shallow embedding of domain-specific languages. *Computer Languages, Systems & Structures*, 44:143–165, 2015. SI: TFP 2011/12.

[25] WOUTER SWIERSTRA. Data types à la carte. *Journal of Functional Programming*, 18(4):423–436, 2008.

[26] Tableau. Tableau prep builder & prep conductor: A self-service data preparation solution, 2021.

[27] Phillip Wadler. The expression problem, Nov 1998.

[28] Jamie Willis, Nicolas Wu, and Matthew Pickering. Staged selective parser combinators. *Proc. ACM Program. Lang.*, 4(ICFP), August 2020.

[29] Nicolas Wu. Yoda: A simple combinator library, 2018.

[30] Brent A. Yorgey, Stephanie Weirich, Julien Cretin, Simon Peyton Jones, Dimitrios Vytiniotis, and José Pedro Magalhães. Giving haskell a promotion. In *Proceedings of the 8th ACM SIGPLAN Workshop on Types in Language Design and Implementation*, TLDI '12, page 53–66, New York, NY, USA, 2012. Association for Computing Machinery.