Dissertation Type: research

DEPARTMENT OF COMPUTER SCIENCE

Circuit: 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.

Saturday 1st 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, Saturday 1st 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 [18]. An example dataflow graph produced by the tool is shown in Figure 2.1

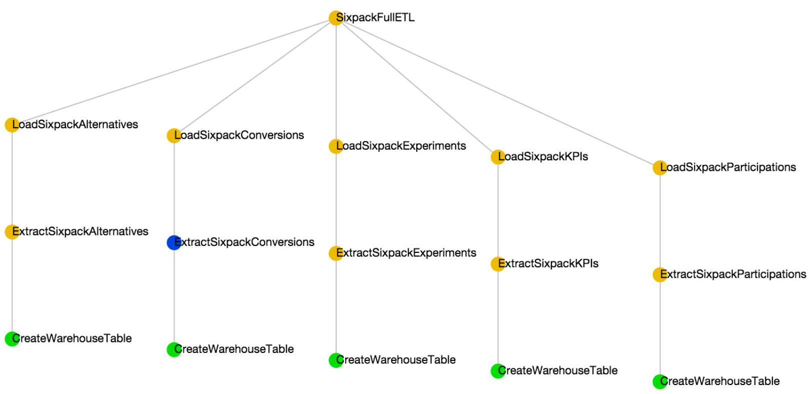


Figure 2.1: Luigi dependency graph [9]

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 [7].

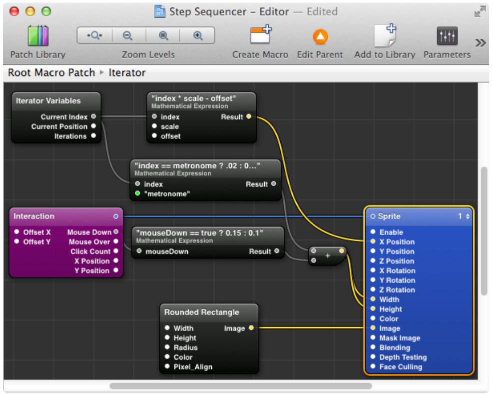


Figure 2.2: Quartz composer [4]

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 [22], that makes programming more accessible to users who do not have programming skills.

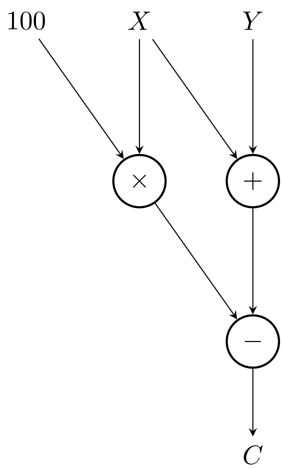
Implicit Parallelism Moore's law states that the number of transistors on a computer chip doubles every two years [15]. This meant that the chips' processing speeds also increased in alignment with Moore's law. However, in recent years this is becoming harder for chip manufacturers to achieve [2]. Therefore, 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.

$$A := 100 \times X$$
$$B := X + Y$$
$$C := A - B$$

(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 [10]. 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.

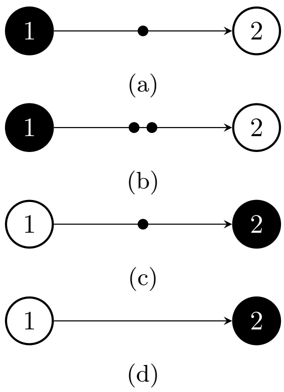


Figure 2.4: A sequence of node firings in a KPN

Parks described a variant of KPNs, called Data Process Networks (DPNs) [13]. 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 [25].

$$\begin{aligned}&\textbf{data}\ \mathsf{Parser_d}\ (\mathsf{a} :: \mathsf{Type})\ \textbf{where}\\ &\quad \mathsf{Satisfy_d} :: (\mathsf{Char} \to \mathsf{Bool}) \to \mathsf{Parser_d}\ \mathsf{Char}\\ &\quad \mathsf{Or_d}\ \ \ \ \ \ :: \mathsf{Parser_d}\ \mathsf{a} \to \mathsf{Parser_d}\ \mathsf{a} \to \mathsf{Parser_d}\ \mathsf{a}\end{aligned}$$

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

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

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

```
parse_d :: Parser_d a → String → [(a, String)]
parse_d (Satisfy_d p) = λcase
  []       → []
  (t : ts') → [(t, ts') | p t]
parse_d (Or_d px py) = λts → parse_d px ts ++ 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 `[('a', "")]`, 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 [24] 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 [20].

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.

```
newtype Parser_s a = Parser_s {parse_s :: String → [(a, String)]}
or_s :: Parser_s a → Parser_s a → Parser_s a
or_s (Parser_s px) (Parser_s py) = Parser_s (λts → px ts ++ py ts)
satisfy_s :: (Char → Bool) → Parser_s Char
satisfy_s p = Parser_s (λcase
  []       → []
  (t : ts') → [(t, ts') | p t])
```

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

```
aorb_s :: Parser_s Char
aorb_s = satisfy_s (≡ 'a') `or_s` satisfy_s (≡ '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:

```
data Expr = Add Expr Expr
          | 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.

```
data ExprF f = AddF f f
             | ValF Int
```

```haskell
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:

```haskell
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 — also known as HFunctor — can be used, which is able to maintain the type indicies [14]. This makes use of $\rightsquigarrow$, which represents a natural transformation 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 [12]. Whereas a functor changes the values inside a structure.

```haskell
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.

```haskell
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.

```haskell
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.

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

The fixed point of ParserF is Parser3.

$$\textbf{type}\ \mathsf{Parser}_{\mathsf{fixed}} = \mathsf{IFix\ ParserF}$$

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.

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

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.

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

The resulting type of icata is f a, therefore the f is a syntactic Functor. This could be IFix ParserF, which would be a transformation to the same structure, possibly applying optimisations to the AST.

is this the right terminology?

2.4 Data types à la carte

When building a DSL one problem that becomes quickly prevalent, the so called *Expression Problem* [23]. 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 [21]. It combines constructors using the co-product of their signatures. This is defined as:

$$\mathbf{data}\ (\mathsf{f} :+: \mathsf{g})\ \mathsf{a} = \mathsf{L}\ (\mathsf{f}\ \mathsf{a}) \mid \mathsf{R}\ (\mathsf{g}\ \mathsf{a})$$

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

$$\begin{aligned}
&\mathbf{instance}\ (\mathsf{Functor}\ \mathsf{f}, \mathsf{Functor}\ \mathsf{g}) \Rightarrow \mathsf{Functor}\ (\mathsf{f} :+: \mathsf{g})\ \mathbf{where}\\
&\quad \mathsf{fmap}\ \mathsf{f}\ (\mathsf{L}\ \mathsf{x}) = \mathsf{L}\ (\mathsf{fmap}\ \mathsf{f}\ \mathsf{x})\\
&\quad \mathsf{fmap}\ \mathsf{f}\ (\mathsf{R}\ \mathsf{y}) = \mathsf{R}\ (\mathsf{fmap}\ \mathsf{f}\ \mathsf{y})
\end{aligned}$$

For each constructor it is possible to define a new data type and a Functor instance specifying where is recurses.

$$\begin{aligned}
&\mathbf{data}\ \mathsf{ValF}_2\ \mathsf{f} = \mathsf{ValF}_2\ \mathsf{Int}\\
&\mathbf{data}\ \mathsf{MulF}_2\ \mathsf{f} = \mathsf{MulF}_2\ \mathsf{f}\ \mathsf{f}\\
&\mathbf{instance}\ \mathsf{Functor}\ \mathsf{ValF}_2\ \mathbf{where}\\
&\quad \mathsf{fmap}\ \mathsf{f}\ (\mathsf{ValF}_2\ \mathsf{x}) = \mathsf{ValF}_2\ \mathsf{x}\\
&\mathbf{instance}\ \mathsf{Functor}\ \mathsf{MulF}_2\ \mathbf{where}\\
&\quad \mathsf{fmap}\ \mathsf{f}\ (\mathsf{MulF}_2\ \mathsf{x}\ \mathsf{y}) = \mathsf{MulF}_2\ (\mathsf{f}\ \mathsf{x})\ (\mathsf{f}\ \mathsf{y})
\end{aligned}$$

By using Fix to tie the recursive knot, the Fix (Val :+: Mul) data type would be isomorphic to the original Expr datatype found in Section 2.3.
One problem that now exist, however, is that it is now rather difficult to create expressions, take a simple example of 12×34.

$$\begin{aligned}
&\mathsf{exampleExpr} :: \mathsf{Fix}\ (\mathsf{ValF}_2 :+: \mathsf{MulF}_2)\\
&\mathsf{exampleExpr} = \mathsf{In}\ (\mathsf{R}\ (\mathsf{MulF}_2\ (\mathsf{In}\ (\mathsf{L}\ (\mathsf{ValF}_2\ 12)))\ (\mathsf{In}\ (\mathsf{L}\ (\mathsf{ValF}_2\ 34)))))
\end{aligned}$$

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 Functors.

$$\begin{aligned}
&\mathbf{class}\ (\mathsf{Functor}\ \mathsf{f}, \mathsf{Functor}\ \mathsf{g}) \Rightarrow \mathsf{f} :\prec: \mathsf{g}\ \mathbf{where}\\
&\quad \mathsf{inj} :: \mathsf{f}\ \mathsf{a} \to \mathsf{g}\ \mathsf{a}\\
&\mathbf{instance}\ \mathsf{Functor}\ \mathsf{f} \Rightarrow \mathsf{f} :\prec: \mathsf{f}\ \mathbf{where}\\
&\quad \mathsf{inj} = \mathsf{id}\\
&\mathbf{instance}\ (\mathsf{Functor}\ \mathsf{f}, \mathsf{Functor}\ \mathsf{g}) \Rightarrow \mathsf{f} :\prec: (\mathsf{f} :+: \mathsf{g})\ \mathbf{where}\\
&\quad \mathsf{inj} = \mathsf{L}\\
&\mathbf{instance}\ (\mathsf{Functor}\ \mathsf{f}, \mathsf{Functor}\ \mathsf{g}, \mathsf{Functor}\ \mathsf{h}, \mathsf{f} :\prec: \mathsf{g}) \Rightarrow \mathsf{f} :\prec: (\mathsf{h} :+: \mathsf{g})\ \mathbf{where}\\
&\quad \mathsf{inj} = \mathsf{R} \cdot \mathsf{inj}
\end{aligned}$$

Using this type class, smart constructors can be defined.

$$\begin{aligned}
&\mathsf{inject} :: (\mathsf{g} :\prec: \mathsf{f}) \Rightarrow \mathsf{g}\ (\mathsf{Fix}\ \mathsf{f}) \to \mathsf{Fix}\ \mathsf{f}\\
&\mathsf{inject} = \mathsf{In} \cdot \mathsf{inj}\\
&\mathsf{val} :: (\mathsf{ValF}_2 :\prec: \mathsf{f}) \Rightarrow \mathsf{Int} \to \mathsf{Fix}\ \mathsf{f}
\end{aligned}$$

```
val x = inject (ValF₂ x)
mul :: (MulF₂ :≺: f) ⇒ Fix f → Fix f → Fix f
mul x y = inject (MulF₂ x y)
```

Expressions can now be built using the constructors, such as val 12 `mul` val 34.

A modular algebra can now be defined that provides an interpretation of this datatype.

```
class Functor f ⇒ EvalAlg f where
  evalAlg :: f Int → Int
instance (EvalAlg f, EvalAlg g) ⇒ EvalAlg (f :+: g) where
  evalAlg (L x) = evalAlg x
  evalAlg (R y) = evalAlg y
instance EvalAlg MulF₂ where
  evalAlg (MulF₂ x y) = x ∗ y
instance EvalAlg ValF₂ where
  evalAlg (ValF₂ x) = x
cata :: Functor f ⇒ (f a → a) → Fix f → a
cata alg (In x) = alg (fmap (cata alg) x)
eval :: EvalAlg f ⇒ Fix f → Int
eval = cata evalAlg
```

One benefit to this approach is that is an interpretation is only needed for expressions that only use MulF and ValF. If a new constructor such as SubF was added to the language and it would never be given to this fold, then it would not require an instance. 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 the experience.

2.5.1 DataKinds Language Extension

Through the use of the DataKinds language extension [26], 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 [6]. 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)
```

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 [17] 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.

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

2.5.4 Summary

Together these features allow for dependently typed programming 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.

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:

- **Easy to build** — This is critical to the success of the language, if it is not simple to use then no one will want to use it. It is important that when defining an application the programmer has a clear understanding of how it will behave.
- **Type-safe** — A feature missing in many Python 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

Tasks are the core construct in the language. 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 a → b
- An external operation - interacting with some external system. For example, calling a terminal command.

Haskell is good for pure functions

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. A data store can be defined as a type class, with two methods fetch and save:

Motivate further why a DataStore needs to exist — prevents the user from reading from a source incorrectly.

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

A DataStore is typed, 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 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.

```haskell
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 could be the aforementioned FileStore:

```haskell
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

A task's type details the type of the input and outputs. It requires two arguments to the constructor, the function that will be invoked and an output DataStore. The constructor makes use of GADTs [16] syntax 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.

```haskell
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

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 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:

```haskell
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
```

This allows for a Task to be combined with others to form a chain. To make this easier to use a chain operator >>> can be defined:

```haskell
(>>>) :: 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.

cite where this comes from?

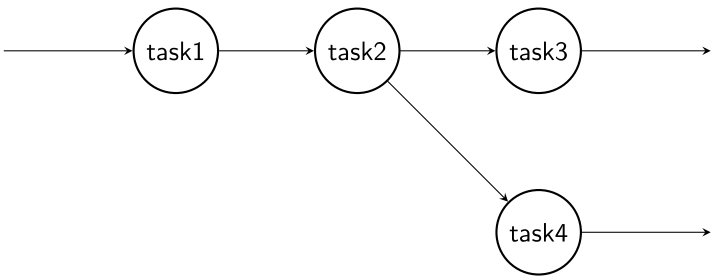
```
data Pipe where
  Pipe :: ∀f a g b.(DataStore f a, DataStore g b) ⇒ Chain f a g b → Pipe
  And  :: Pipe → Pipe → Pipe
```

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).

There is, however, one 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. Similarly, if a user wanted to use the same task multiple times, it would not be possible to differentiate between them.

Process Identifiers (PIDs) This is where the concept of Process Identifiers (PIDs) are useful. A Chain can be modified so that instead of storing a Task it instead stores a PID. The PID data type can make use of phantom type parameters, to retain the same information as a Task, whilst storing just an Int that can be used to identify it.

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

```
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 however leaves a key question, how do Tasks get mapped to PIDs. This can be done by employing the State monad. This state stores a map from PID to task and a counter for PIDs. As Tasks each have a different type again a new datatype is required to use existential types, so that just one type is stored in the map. By creating a type alias for the State monad, the Workflow monad can now be used.

```
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
```

There is only one operation that is defined in the monad — 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

Easy to Build The concept of chains are easy for a user to grapple with. Chains can be any length and represent paths along a dataflow graph. The chains can be any length that the user requires. This gives them the choice of how to structure the program. In one case they could specify a minimal number of chains that describe the dataflow graph. However, another approach from the user could be to just focus dependencies between each tasks, and specifying a chain for each edge in the dataflow graph.

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.

3.4 Circuit

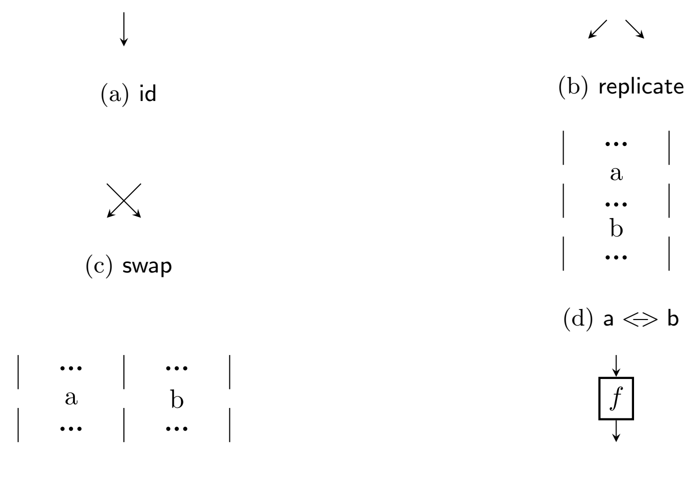
This approach was inspired by the parallel prefix circuits as described by Hinze [8]. It uses constructors similar to those used by Hinze to create a circuit that represents the Directed Acyclic Graph (DAG), used in the dataflow. The constructors seen in Figure 3.2 represent the behaviour of edges in a graph.

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:

```
Circuit (inputsStorageTypes  :: [Type → Type]) (inputsTypes  :: [Type]) (inputsApplied  :: [Type])
        (outputsStorageTypes :: [Type → Type]) (outputsTypes :: [Type]) (outputsApplied :: [Type])
        (nInputs :: Nat)
```

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 [26], to use type-level lists and natural numbers. Each parameter represents a certain piece of information needed to construct a circuit:



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

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

- inputStorageTypes is a type-list of storage types, for example '[VariableStore, CSVStore].
- inputTypes 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, outputTypes 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.

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, using phantom type parameters [3]. 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 of type f a and it returns that same value. Some more interesting examples would be the swap and replicate:

```
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
```

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.

All three of these constructors are leaf nodes in the AST. To be able to make use of them they need to be combined in some way. To do this two new 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.

might be a bit hand wavy description...?

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 [5] is created that is able to apply the two type lists together. This type family pairwise applies a list of types storing 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. To do this an append type family [11] can be used:

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

This type family makes use of the language extension PolyKinds [26] 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. Through types it enforces that the output of the top circuit is the same as the input to the bottom circuit.

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

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.

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

3.4.2 Combined DataStores

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, it allows for them to also be defined over type lists, not just a single type. Combined DataStores make it easier for tasks to fetch from multiple inputs. Users will just have to call a single fetch' function, rather than multiple.

To be able to define DataStore′, heterogeneous lists [11] are needed — specifically three different forms. HList is as defined by TODO, 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. Their definitions are:

```
data HList (xs :: [Type]) where
  HCons :: x → HList xs → HList (x ': xs)
  HNil  :: HList '[]
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
  save'  :: HList' fs as → HList as → IOList (Apply fs as)
```

However, it would be cumbersome to ask the user to define an instance of DataStore′ for every possible combination of data stores. Instead, it is possible to make use of the previous DataStore type class. To do this instances can be defined for DataStore′ that make use of the existing DataStore instances:

```
instance {-# OVERLAPPING #-} (DataStore f a) => DataStore' '[f] '[a] where
  fetch' (HCons' x   HNil')                  = IOCons (fetch x)    IONil
  save'  (HCons' ref HNil') (HCons x HNil)   = IOCons (save ref x) IONil
instance (DataStore f a, DataStore' fs as) => DataStore' (f ': fs) (a ': as) where
  fetch' (HCons' x   xs)             = IOCons (fetch uuid x) (fetch' xs)
  save'  (HCons' ref rs) (HCons x xs) = IOCons (save ref x)  (save' rs xs)
```

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 DataStores.

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.

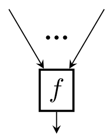


Figure 3.3: A graphical representation of a task with multiple dependencies

To support this a modification can be 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.

What is Length???

```
task :: (DataStore' fs as, DataStore g b)
  => (HList' fs as -> g b -> IO (g b))
  -> g b
  -> Circuit fs as (Apply fs as) '[g] '[b] '[g b] (Length 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:

```
multiInputTask :: (DataStore' fs as, DataStore g b)
  => (HList as -> b)
  -> g b
  -> Circuit fs as (Apply fs as) '[g] '[b] '[g b] (Length fs)
functionTask :: (DataStore f a, DataStore g b)
  => (a -> b)
  -> g b
  -> Circuit '[f] '[a] '[f a] '[g] '[b] '[g b] N1
```

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 a → 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.

$$
\begin{aligned}
&\mathsf{mapC} :: (\mathsf{DataStore}'\;{}'[\mathsf{f}]\;{}'[[\mathsf{a}]], \mathsf{DataStore}\;\mathsf{g}\;[\mathsf{b}]) \\
&\quad \Rightarrow \mathsf{Circuit}\;{}'[\mathsf{VariableStore}]\;{}'[\mathsf{a}]\;{}'[\mathsf{VariableStore}\;\mathsf{a}]\;{}'[\mathsf{VariableStore}]\;{}'[\mathsf{b}]\;{}'[\mathsf{VariableStore}\;\mathsf{b}]\;\mathsf{N1} \\
&\quad \rightarrow \mathsf{g}\;[\mathsf{b}] \\
&\quad \rightarrow \mathsf{Circuit}\;{}'[\mathsf{f}]\;{}'[[\mathsf{a}]]\;{}'[\mathsf{f}\;[\mathsf{a}]]\;{}'[\mathsf{g}]\;{}'[[\mathsf{b}]]\;{}'[\mathsf{g}\;[\mathsf{b}]]\;\mathsf{N1}
\end{aligned}
$$

Graphical representation of this?

3.4.5 Completeness

something about the stuff Alex said

monadic resource theories.

3.4.6 Evaluation

Easy to Build A Circuit focuses on the transformations that are made to edges on a graph. This can be beneficial to the user as it is the edges in a dataflow diagram that encode dependencies between tasks. Although circuits may initially appear complex, there is a relatively simple process that can be used to construct them. By hand-drawing a dataflow diagram, a circuit can always be constructed that closely mirrors this diagram. This means that the user can easily visualise what is happening inside a circuit. In fact it could be possible to pretty print a circuit to recreate this 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

- Typesafe
- Parallel
- Competitive Speed
- Failure Tolerance

4.2 Circuit AST

How have i modified a la carte to work with ifunctors. then give a few examples and maybe one of the smart constructors that injects the L's and R's

4.3 Network

4.3.1 Network Typeclass

Interaction with Network

4.4 Translation

4.4.1 Steps of translation

icataM

4.4.2 UUIDS

Why are they needed?

How are they added?

4.5 Failure in the Process Network

Why?

4.5.1 Maybe Monad

No error messages

4.5.2 Except Monad

based on Either

Chapter 5

Examples

5.1 How to build a Circuit

5.2 Song Data Aggregation

5.3 lhs2TeX Build System

5.4 Types saving the day

Consider an example shown in the docs for Luigi [19], 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 5.1, 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 5.1: 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 5.2. 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 (foldr f []) (FileStore "count.txt")
  where
    f word cs = (concat [word, ":", show (length word)]) : cs

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

Figure 5.2: A Broken Circuit Example

Chapter 6

Critical Evaluation

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

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

[4] 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.

[5] 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.

[6] 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.

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

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

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

[10] 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.

[11] 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.

[12] 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.

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

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

[15] 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.

[16] 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.

[17] 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.

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

[19] Spotify. Tasks, Apr 2020.

[20] 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.

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

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

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

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

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

[26] 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.