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.

Monday 19th April, 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, Monday 19th April, 2021

Contents

Todo list

Acknowledgements

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

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. Edges connect these nodes together, and define the flow of information.

this feels a little light on detail

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.

Example - Quartz Composer Apple developed a tool included in XCode, named Quartz Composer, which is a node-based visual programming language [1]. It allows for quick development of programs that process and render graphical data. By using visual programming it allows the user to build programs, without 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[3].

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 [10]. This makes programming more accessible to users who do not have programming skills.

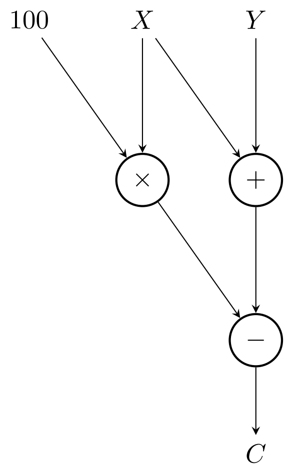
Implicit Parallelism Moore's law states that the number of transistors on a computer chip doubles every two years [8]. This meant that the chips processing speeds also increased in alignment with Moore's law. However, in recent years this is becoming harder for chip manufacturers to achieve [2]. Therefore, 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, is visible in Figure 2.1. 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.

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

(a)



(b)

Figure 2.1: An example dataflow and its imperative approach.

2.1.3 Kahn Process Networks

A method introduced by Gilles Kahn, called Kahn Process Networks (KPN) realised this concept through the use of threads and unbounded FIFO queues [4]. A node in the dataflow becomes a thread in the process network. These 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. A requirement of KPNs is that a thread is suspended if it attempts to fetch a value from an empty queue. It is not possible for a process to test for the presence of data in a queue.

Parks described a variant of KPNs, called Data Processing networks [6]. 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 Domain Specific Language (DSL) is a programming language unit 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. 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 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 embedded DSL, can be implemented using two differing techniques: shallow and deep embeddings.

Add something about why embedded DSLs are used in Haskell

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 an eval function. Consider the example of a minimal non-deterministic parser combinator library [13].

```
data Parser (a :: Type) where
  Satisfy :: (Char → Bool) → Parser Char
  Or      :: Parser a → Parser a → Parser a
```

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

```
aorb :: Parser Char
aorb = Satisfy (≡ 'a') `Or` Satisfy (≡ 'b')
```

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

```
parse :: Parser a → String → [(a, String)]
parse (Satisfy p) = λcase
  []        → []
  (t : ts') → [(t, ts') | p t]
parse (Or px py) = λts → parse px ts ++ parse py ts
```

The program can then be evaluated by the parse function. For example, parse aorb "a" evaluates to , and parse aorb "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 makes use of such techniques to create optimised parsers [12]. 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 [9].

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 Parser₂ a = Parser₂ {parse₂ :: String → [(a, String)]}
or :: Parser₂ a → Parser₂ a → Parser₂ a
or (Parser₂ px) (Parser₂ py) = Parser₂ (λts → px ts ++ py ts)
satisfy :: (Char → Bool) → Parser₂ Char
satisfy p = Parser₂ (λcase
  []        → []
  (t : ts') → [(t, ts') | p t])
```

The same aorb parser can be created directly from these functions, avoiding the need for an intermediate AST.

```
aorb₂ :: Parser₂ Char
aorb₂ = satisfy (≡ 'a') `or` satisfy (≡ '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 - you cannot inspect the structure. 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. There is, however, one problem: a Functor expressing the 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 the type of a parser. Instead a type class called IFunctor can be used, which is able to maintain the type indicies [7]. This makes use of ⇝, 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 [5].

```
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

```
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 Fix f = In (f (Fix f))
newtype IFix iF a = IIn (iF (IFix iF) a)
```

The fixed point of ParserF is Parser_3.

```
type Parser₃ = IFix ParserF
```

In a deep embedding, the AST is traversed and modified to make optimisations, however, it may not be the best representation when evaluating it. This means that it is usually 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 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 layer of the datatype.

```
icata :: IFunctor iF ⇒ (iF f ⇝ f) → IFix iF ⇝ f
icata alg (IIn x) = alg (imap (icata alg) x)
```

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

2.4 Data types à la carte

When building a DSL one problem that becomes quickly prevalent, the so called *Expression Problem* [11]. 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. It combines constructors using the coproduct of their signatures. This is defined as,

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

For each constructor it is possible to define a new data type.

```
data Val f = Val Int
data Mul f = Mul f f
```

By using Fix to tie the recursive knot, the Fix (Val :+: Mul) data type would be isomorphic to a standard Expr data type.

```
data Expr = Add Expr Expr
  | Val Int
```

One problem that now exist, however, is that it is now rather difficult to create expressions, take a simple example of 12×34.

```
exampleExpr :: Fix (Val :+: Mul)
exampleExpr = In (R (Mul (In (L (Val 12))) (In (L (Val 34)))))
```

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.

```
class (Functor f, Functor g) ⇒ f :≺: g where
  inj :: f a → g a
instance Functor f ⇒ f :≺: f where
  inj = id
instance (Functor f, Functor g) ⇒ f :≺: (f :+: g) where
  inj = L
instance (Functor f, Functor g, Functor h, f :≺: g) ⇒ f :≺: (h :+: g) where
  inj = R · inj
```

Using this type class, smart constructors can be defined.

```
inject :: (g :≺: f) ⇒ g (Fix f) → Fix f
inject = In · inj
val :: (Val :≺: f) ⇒ Int → Expr f
val x = inject (Val x)
(∗) :: (Mul :≺: f) ⇒ Fix f → Fix f → Fix f
x ∗ y = inject (Mul x y)
```

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

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

Add some refs

Through the use of the DataKinds language extension, all data types are promoted to also be kinds and their constructors to be type constructors. When constructors are promoted to type constructors, they are prefixed with a '. For example Zero 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. One case where this could be is getting the head of a list. If you attempt to get 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 :: Vector (Succ n) a -> a
safeHead (Cons x _) = x
```

2.5.2 Singletons

Add some refs

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
```

It turns out this is possible through the use of singletons. A singleton in Haskell is a type that has just one inhabitant. 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

Add some refs

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 become useful, they allow for the definition of functions on types. The ideal type signature for appending two vectors together would be,

```
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)
```

2.6 Type Families

- What are they?
- DataKinds
- Examples
- What is is?
- Singletons, why they needed, examples, using with typefamilies.

Chapter 3

Project Execution

Chapter 4

Critical Evaluation

Chapter 5

Conclusion

Bibliography

[1] Quartz composer user guide, Jul 2007.

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

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

[4] 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.

[5] 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.

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

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

[8] 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.

[9] 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.

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

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

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

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