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.

Thursday 15th 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, Thursday 15th April, 2021

Contents

1 Introduction **1**

2 Background **3**
2.1 Dataflow Programming 3
2.2 Domain Specific Languages (DSLs) 4
2.3 Higher Order Functors 5
2.4 Type Families 6
2.5 Dependently Typed Programming 6

3 Project Execution **7**

4 Critical Evaluation **9**

5 Conclusion **11**

Todo list

this feels a little light on detail 3
Add something about why embedded DSLs are used in Haskell 4
reads dodgy 5

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 [7]. 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 [6]. 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.

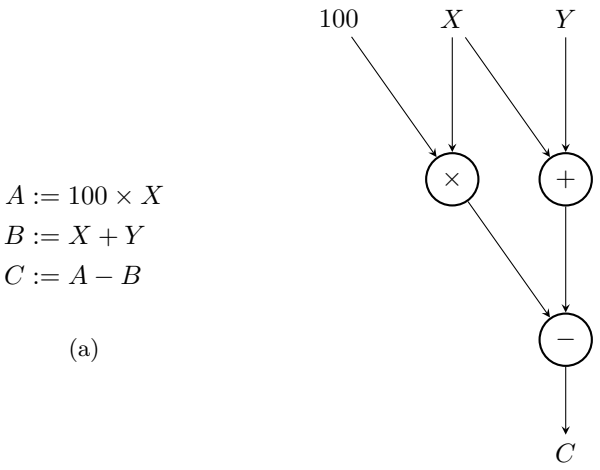

Figure 2.1: An example dataflow and its imperative approach.

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 [5]. 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 [8].

```
data Parser₂ (a :: Type) where
  Satisfy :: (Char → Bool) → Parser₂ Char
  Or      :: Parser₂ a     → Parser₂ a → Parser₂ a
```

The same aorb parser can be created by creating an AST. reads dodgy

```
aorb₂ :: Parser₂ Char
aorb₂ = Satisfy (≡ 'a') `Or` Satisfy (≡ 'b')
```

However, this parser does not have any semantics, therefore this needs to be provided by the evaluation function parse_2.

```
parse₂ :: Parser₂ a → String → [(a, String)]
parse₂ (Satisfy p) = λcase
  []       → []
  (t : ts') → [(t, ts') | p t]
parse₂ (Or px py) = λts → parse₂ px ts ++ parse₂ py ts
```

A key benefit for deep embeddings is that the structure can be inspected, and then modified to optimise the user code. 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.

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 a = Parser {parse :: String → [(a, String)]}

or :: Parser a → Parser a → Parser a
or (Parser px) (Parser py) = Parser (λts → px ts ++ py ts)

satisfy :: (Char → Bool) → Parser Char
satisfy p = Parser (λcase
  []       → []
  (t : ts') → [(t, ts') | p t])
```

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')
```

The program can then be evaluated by the parse function. For example, `parse aorb "a"` evaluates to `[('a', "")]`, and `parse aorb "c"` evaluates to `[]`.

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

Introduce the need for them... folding typed ASTs to provide syntax.

- IFunctors, imap, natural transformation
- Maybe drop some cat theory diagrams
- IFix
- Their use for DSL development, icata, small example.

2.4 Type Families

- What are they?
- DataKinds
- Examples

2.5 Dependently Typed Programming

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

[6] 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.

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

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