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

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

programming paradigm that represents applications as a DAG (like a dataflow diagram). nodes with inputs and outputs. nodes are sources, sinks or processing nodes. nodes connected by directed edges which define the flow of information

2.1.1 The Benefits

Visual visual programming language, easier for the end user to visualise what is happening.

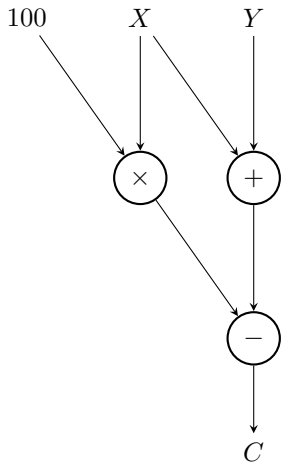
Implicit Parallelism implicit parallelism [1], each node is pure and has no side effects. no data dependencies. parallelism is now more important because of multicore cpus and the need to process large amounts of data, that can benefit from parallel processing.

2.1.2 Dataflow Diagrams

In the traditional imperative approach, the code written will be sequential, with each line executed one after another. An example is visible in Figure 2.1a. However, in

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

(a)



(b)

Figure 2.1: an example 2.1b

Give some example diagrams for a data flow

Give a simple expression style program and a dataflow equivalent.

Batch Processing What is it? and where is it used?
An example?

Stream Processing What is it and where is it used?
An example?

Batch vs Stream Comparison of features, discuss how dataflow programming can be used for both.

2.1.3 Kahn Process Networks

What are they?
How can they be used to model a Kahn process network? Maybe give an example diagram
Discuss the specific case i am using, where the firing of a node will always pop 1 from the input tape.

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 Shallow Embeddings

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. Consider the example of a minimal non-deterministic parser combinator library [2].

```
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 `[]`.

Add the advantages of shallow embeddings.

2.2.2 Deep Embeddings

Alternatively, a deep embedding can be used to represent a DSL. This 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. Again a simple parser example can be considered.

```
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
```

Add the advantages of deep embeddings.

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] Wesley M. Johnston, J. R. Paul Hanna, and Richard J. Millar. Advances in dataflow programming languages. *ACM Comput. Surv.*, 36(1):1–34, March 2004.

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