Transparent Synchronous Dataflow

A functional paradigm for systems modelling and optimisation

by

Steven Wai Tak Cheung

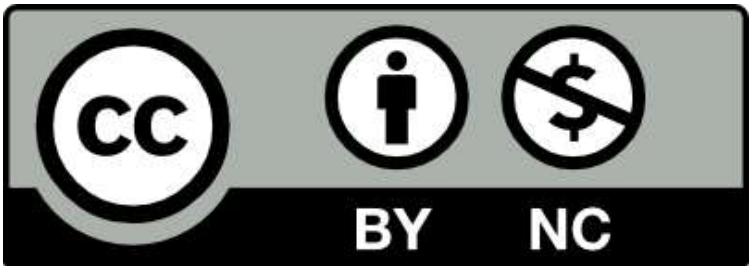
A thesis presented for the degree of
Doctor of Philosophy



School of Computer Science
College of Engineering and Physical Sciences
University of Birmingham
Sep 2020

UNIVERSITY OF BIRMINGHAM

University of Birmingham Research Archive
e-theses repository



This unpublished thesis/dissertation is under a Creative Commons Attribution-NonCommercial 4.0 International (CC BY-NC 4.0) licence.

You are free to:

Share — copy and redistribute the material in any medium or format

Adapt — remix, transform, and build upon the material

The licensor cannot revoke these freedoms as long as you follow the license terms.

Under the following terms:



Attribution — You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.



NonCommercial — You may not use the material for commercial purposes.

No additional restrictions — You may not apply legal terms or technological measures that legally restrict others from doing anything the license permits.

Notices:

You do not have to comply with the license for elements of the material in the public domain or where your use is permitted by an applicable exception or limitation.

No warranties are given. The license may not give you all of the permissions necessary for your intended use. For example, other rights such as publicity, privacy, or moral rights may limit how you use the material.

Unless otherwise stated, any material in this thesis/dissertation that is cited to a third-party source is not included in the terms of this licence. Please refer to the original source(s) for licencing conditions of any quotes, images or other material cited to a third party.

Abstract

System modelling is the use of mathematical formalisms to model real world systems for the purpose of analysis, simulation and prediction. One of the most common ways to model a system is to create the dataflow among its various components. There are two main approaches on how dataflow graphs are constructed in these system modelling frameworks: *'define-and-run'* vs *'define-by-run'*. The former approach first creates a dataflow graph and then executes it by pushing data into it. The latter however constructs the graph while computing with data on-the-fly. 'Define-and-run' is usually more efficient because many graph optimisations can be applied; 'Define-by-run' however handles dynamic models better. This thesis aims to develop a new functional paradigm for systems modelling and optimisation that exhibits properties of both approaches where dataflow graphs are dynamic but efficient.

We propose a new functional language, namely *transparent synchronous dataflow* (TSD), where dataflow graphs are constructed *transparently* with imperative commands to manipulate them explicitly; together with a synchronous mode of *change propagation*. The semantics of the language is designed on top of an unconventional graph abstract machine, Dynamic Geometry of Interaction Machine (DGoIM), which is natural for manipulating dataflow graphs. By using this semantics, the language is proved to be sound and efficient. Several experimental implementations were also created, including a native compiler for DGoIM and OCaml implementations for TSD.

For my wife Jessily Wai Yau Ying; and my parents Siu Sun Cheung and Miu Sim Kwan

Acknowledgements

I would like to express my deepest gratitude towards my supervisor Dan R. Ghica who is always keen on helping his students to great extent. His timely advice and continuous support throughout the years have been crucial for me to complete my work.

I am also grateful to the research group here in Birmingham, especially my thesis group members, Martín Hötzel Escardó and Rami Bahsoon for their suggestions and guidance. Special thanks to Martin who introduced me to the world of theoretical computer science and encouraged me to pursue a doctorate. Many thanks to Koko Muroya, a great collaborator and a friend, who has offered tremendous help.

I am also thankful to the examiners, Andrzej Murawski and Vincent Rahli for their time and valuable critiques which help to improve my work.

Thanks to the University of Birmingham who has offered me a studentship to support my study. Also thanks to the Vocational Training Council and the Alister Harvey Foundation Scholarship who sponsored my bachelor study which sparked this incredible journey in England.

Finally, to all my friends and family, especially my wife Jessily Ying, for all the support you have given me in all these years.

Contents

1 Introduction **1**

1.1 Background 1

1.2 Problem statements 4

1.2.1 Straight-line code construction 4

1.2.2 Symmetric optimisation 7

1.2.3 A unified semantics 9

1.3 Contributions 15

1.4 Thesis outline 17

2 Related Work **19**

2.1 Deep learning frameworks 19

2.1.1 Define-and-run 20

2.1.2 Define-by-run 28

2.1.3 Idealised TensorFlow 31

2.2 Reactive programming languages 32

2.2.1 Synchronous dataflow languages 33

2.2.2 Self-adjusting computation 36

3 Transparent Synchronous Dataflow **39**

3.1 The dataflow core 39

3.2 Extensions for parameters optimisation 43
3.3 Syntactic name-polymorphism 44
3.4 Abstract machine semantics 46

4 Programming in TSD **56**

4.1 Linear regression[1] 56
4.1.1 Gradient descent[2] 56
4.1.2 Using gradient descent 58
4.1.3 Parallelisation and pipelines 61
4.2 Recurrent neural networks[3] 65
4.3 Eco-sphere[4] 68
4.4 Optimal road design[5] 70
4.5 Sieve of Eratosthenes[6] 72

5 The TSD calculus **75**

5.1 Formal syntax 75
5.1.1 TSD 75
5.1.2 simTSD 79
5.1.3 From TSD to simTSD 80
5.1.4 Soundness of transformation 82
5.2 Formal semantics 86
5.2.1 Graphs and translation 86
5.2.2 Machine states 88
5.2.3 Transitions 92

6 Theoretical Results **109**

6.1 Conventions 113
6.2 Dataflow graphs 113

6.3 Name equivalence . . . 115
6.4 Validity . . . 117
6.4.1 Valid graphs . . . 117
6.4.2 Valid states . . . 120
6.5 Determinism . . . 129
6.6 Expansion of Contractions . . . 133
6.6.1 One-step simulation . . . 133
6.6.2 Expanding contractions . . . 136
6.7 Termination of Recursion-free Programs . . . 139
6.8 Type Soundness . . . 150
6.9 Symmetry . . . 152
6.10 Graph equivalences . . . 155
6.10.1 U-simulation . . . 156
6.10.2 Rewiring . . . 157
6.10.3 Copying . . . 159
6.10.4 Engulfing . . . 160
6.10.5 Beta-equivalence . . . 162
6.11 Complexity of transitions . . . 162

7 Experimental Implementations **165**

7.1 Native compiler for DGoIM . . . 165
7.1.1 Modified abstract machine . . . 167
7.1.2 Code generation . . . 169
7.1.3 Evaluation . . . 172
7.2 The dataflow core as an OCaml library . . . 173
7.2.1 Incremental version . . . 176
7.2.2 Benchmarks . . . 176

7.3 Extensions for parameters optimisation 180

8 Evaluation **185**

8.1 Straight-line code construction 185

8.2 Symmetric optimisation 187

8.3 A unified semantics 189

8.4 Comparisons with other languages 190

8.4.1 Deep learning frameworks 191

8.4.2 Reactive programming languages 195

9 Conclusion **197**

9.1 Future Work 198

List of Figures

1.1 FIR filter[7] 2

1.2 Linear regression, example 8

1.3 Evaluation of $(\lambda x.x)\ (1 + 2)$ 15

2.1 Various deep learning frameworks 20

2.2 Model in Caffe 21

2.3 Linear regression in Torch 22

2.4 Dataflow graphs in Torch 24

2.5 Linear regression in Theano 25

2.6 Linear regression in early TensorFlow 26

2.7 Linear regression in PyTorch 28

2.8 Static subgraph in Chainer 29

2.9 Linear regression in TF Eager 30

3.1 Informal example for state machines 41

3.2 Basic example $(\lambda x.x + x + 1)\ 2$ 47

3.3 Dataflow graph $(\lambda x.\lambda y.(y + y) + (x + x))$ (`deref` (`ref` 0)) 2 49

3.4 Dataflow graph $(\lambda x.\lambda y.(y + y) + (x + x))$ (`deref` (`ref` 0)) 2 50

3.5 Linking 51

3.6 Step propagation 52

3.7 Step propagation with cycle 53

3.8 Fusion 54

4.1 Gradient descent 57
4.2 Numerical gradients graph 58
4.3 Linear regression 58
4.4 Graphical model of linear regression 59
4.5 Numerical gradients graph for the linear regression example 60
4.6 Full example of linear regression with batch gradient descent 64
4.7 Helper functions for creating neural networks 65
4.8 A LSTM unit[8] 66
4.9 LSTM in TSD 67
4.10 Eco-sphere model 68
4.11 Ecosphere: the model 69
4.12 Shrimp population evolutions 70
4.13 A 3-segment road model 70
4.14 Optimised road (in kilometers) 71
4.15 Sieve of Eratosthenes in TSD 72
4.16 Dataflow graph expansion for primes 74
4.17 Sieve of Eratosthenes in Lucid Synchrone 74

5.1 Typing rules of TSD 77
5.2 Typing rules of simTSD 80
5.3 Transformation from TSD to simTSD 81
5.4 Connection of edges 87
5.5 Conventions for multiple nodes and edges 87
5.6 Translation of TSD types 87
5.7 Translation of TSD terms (1) 89
5.8 Translation of TSD terms (2) 90

5.9 Translation of $(\lambda x.x + x)(1 + 2)$ 90
5.10 Port labels 94
5.11 Dataflow transitions 95
5.12 Pass rules of evaluation transitions 99
5.13 Computation rewrites 101
5.14 Computation rewrites for TSDextension 103
5.15 Fusion rewrite 104
5.16 Example of fuse-rule 106
5.17 Copying rewrites 106

6.1 Example of `foldp` 111
6.2 δ-rewrite on F_a-box 118

7.1 Syntax tree 166
7.2 Translation to graphs 167
7.4 Beta-rewiring in native compile 168
7.3 $\lambda x.x + x$ 168
7.5 Copying in native compile 169
7.6 Code generation (Var) 170
7.7 Testing program 172
7.8 Example program in the OCaml implementation 175
7.9 Difference between fold-sum and fold-sum 2 179

8.1 Various deep learning frameworks, revisited 191

List of Tables

5.1 Primitive operations 76

7.1 Native compiler vs OCaml 172

7.2 Benchmarks against RML and Zélus 178

7.3 Benchmarks against JS incremental 179

8.1 Lustre primitives using TSD 196

Chapter 1

Introduction

This thesis aims to develop a new functional paradigm for systems modelling and optimisation, namely *transparent synchronous dataflow*, where models are represented by dataflow graphs constructed *transparently*, together with a *synchronous* mode of change propagation.

1.1 Background

Systems modelling System modelling is the use of mathematical formalisms to model real world systems for the purpose of analysis, simulation and prediction. For example digital controllers such as a traffic light controller can be modelled as a discrete event system [74], e.g. automata, while planetary motions can be described as continuous time system modelled by differential equations, in this particular case Kepler's equations [9]. The chosen models should have the right level of abstraction, which is simple enough to understand and to be examined while keeping all the essential details.

One of the most common ways to model a system is to create the dataflow among its various components [52, 15]. It is based on the simple principle that systems can be represented as a directed graph that reacts to changes in the inputs and propagates through itself automatically. A typical example is a finite impulse response (FIR) filter frequently used in

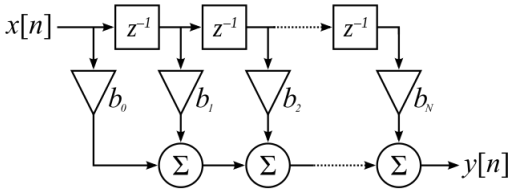


Figure 1.1: FIR filter[1]

signal processing. It can be defined by the following equation:

$$\begin{aligned} y(n) &= b_0 \cdot x(n) + b_1 \cdot x(n-1) + \cdots + b_N \cdot x(n-N) \\ &= \sum_{i=0}^{N} b_i \cdot x(n-i) \end{aligned}$$

where $x(n)$ and $y(n)$ are the input and output signals, N is the filter order indicating the number of previous time instances that are used, and b_i is the impulse response at the ith instant of the filter. In other words, the output of a FIR filter is the sum of the N latest inputs each multiplied by some coefficients. The corresponding dataflow graph is shown in Fig. 1.1 where z^{-1} is a delay operator, and the triangle node represents multiplication by some coefficients. It is common to represent operations as nodes or blocks with a set of ports that can be divided into the inputs or outputs. These nodes are then connected by directed edges indicating the flow of information between them.

Using dataflow graphs enables a number of features [52]. Firstly, for every node in a graph, once its inputs are ready, it can then be executed automatically without having to synchronise with other nodes. As such, dataflow graphs are inherently parallel, and this parallelism is often provided transparently, meaning that much of the lower-level mechanisms are hidden from programmers. Secondly, since the execution of dataflow graphs is parallel in nature, it is possible to distribute different parts of the graph across multiple devices attached to different machines. Thirdly, various graph optimisations, such as removal of unreachable nodes or identity nodes, are applicable. Finally, dataflow graphs facilitate an easy and efficient implementation of *automatic differentiation* (AD) [43]. Many of the conventional AD

[1]Extracted from `https://en.wikipedia.org/wiki/Finite_impulse_response#/media/File:FIR_Filter.svg`

techniques such as *back-propagation* apply to so-called 'straight-line code' which corresponds to ground-type recursion-free pure functional code [44]. Since models in general are functions of type $\mathbb{R}^n \to \mathbb{R}^m$, therefore in most cases straight-line code is expressive enough. However, there are some notable exceptions to the rule, for example LSTM neural networks which have a stateful model requiring a special technique called *back-propagation-through-time* [17, 53]. On the other hand, applying AD to higher order languages with or without effects remains an active area of research without definitive solutions [11].

System modelling languages System modelling languages and toolkits have been a broad area of research both commercial and academic. There have been many different approaches to language design: imperative vs functional vs objected-oriented, synchronous vs asynchronous and visual vs textual. For example Simulink [33] and LabView [14] are visual languages in which dataflow models can be created graphically by assembling block diagrams, while Modelica [37] is an object-oriented language where models are created as reusable objects encapsulating the equations of a particular system. Other examples include the various reactive programming languages which are concerned with applications that involve time-varying values and propagation of changes [10]. The concept of dataflow graph is also brought into programming frameworks for deep learning such as TensorFlow [1] and PyTorch [68].

The common design of system modelling languages is 'modal' in the sense that they distinguish two modes of execution: a model-building mode and a model-execution mode. Simulink [33], earlier Tensorflow [1], or Lucid Synchrone [21] are examples of such design. The model-building mode constructs the dataflow graph while the model-execution mode evaluates the model. For example in spreadsheet systems, the model-building mode is simply the manual manipulation of the spreadsheet itself, while the model-execution mode happens when the value of a spreadsheet cell is modified, triggering the propagation of changes in a fully transparent manner. Some of these languages, e.g. TensorFlow, are embedded into a more expressive host language, e.g. Python; while some are stand-alone tools or frameworks,

e.g. Modelica or Simulink.

In addition to constructing and executing models, some frameworks also have built-in support for optimisation which automatically tunes the control parameters to fit into a set of predefined constraints. For example Simulink, LabView and Zélus all come with optimisation modules that include differential equation solvers (Optimisation Toolbox[2], Simulink Design Optimization[3] and SIM Optimal Design VI[4]). Modelica also has extension (Optimical [5]) which supports dynamic optimisation. Deep learning frameworks such as TensorFlow and PyTorch also come with rich optimisation libraries.

1.2 Problem statements

1.2.1 Straight-line code construction

Automatic differentiation is the heart of modern deep learning frameworks. Conventional AD techniques mostly work on straight-line codes. For simple models, writing straight-line code directly is not a problem. However, in practice constructing huge models with complicated topology, for instance a neural network with tens of thousands of parameters, requires the model to be created using a more expressive programming language. In that way, the model is then a straight-line code only at runtime.

The straight-line code model can be constructed in meta-programming style in platforms such as early TensorFlow. Let us consider a simple linear regression example written in TensorFlow [25]:

```
import tensorflow as tf

x = tf.placeholder(tf.float32) # creates an input node x
W = tf.Variable([2.]) # creates the weight
b = tf.Variable([1.]) # creates the bias
y_pred = tf.add(tf.multiply(x, W), b) # creates a graph for 'y_pred = x * W + b'
```

[2] https://uk.mathworks.com/products/optimization.html
[3] https://uk.mathworks.com/products/sl-design-optimization.html
[4] http://zone.ni.com/reference/en-XX/help/371894J-01/lvsim/sim_optimal_design_vi/

The function `tf.placeholder` creates the input node, `tf.Variable` creates a model parameter; and `tf.add` and `tf.multiply` create the operator nodes for addition and multiplication respectively. This style of graph construction is similar to building a syntax tree, which is sometimes called meta-programming. In order to execute the graph, the following code is needed:

```
init = tf.global_variables_initializer() # parameters initialiser
sess = tf.Session() # execution engine
sess.run(init) # initialises model parameters
print(sess.run(y, {x:[1, 2, 3, 4, 5, 6]})) # => [3,  5,  7, 9, 11, 13]
```

Firstly, an initialiser for parameters `init` is created. Secondly, a session object `sess = tf.Session()` is instantiated, and model parameters are initialised by `sess.run(init)`. Finally, `sess.run(y, x:[1,2,3,4,5,6])` executes the graph by feeding data into the input node `x`.

On the other hand, if our goal is to train the model then we need another placeholder for the actual output, a loss function that can be minimised, an optimiser and a graph that calculates the gradients. For example, in the code below, we chose the gradient descent optimiser. The learning rate specifies how quickly the model is adapted to the training data.

```
# creates the input node for actual value of 'y'
y = tf.placeholder(tf.float32)
# mean squared error
loss = tf.reduce_mean(tf.square(y - y_pred))

# gradient descent optimiser
optimizer = tf.train.GradientDescentOptimizer(learning_rate)
# creates the graph for automatic differentiating the loss function
train = optimizer.minimize(loss)

# training data
feed_dict = {x: x_batch, y: y_batch}

init = tf.global_variables_initializer()
sess = tf.Session()
sess.run(init)
for i in range(100):
  # calculuates the gradients and updates parameters
  sess.run(train, feed_dict)
```

Therefore, whether we want to execute or to train the model in TensorFlow, we first create the dataflow graph and then use a `session` object to push the data through the graph and obtain the results. This style of graph construction and execution is often referred to as *'define-and-run'* or *'modal'*. Furthermore, graphs in TensorFlow are static, meaning that they cannot be changed during execution. Although this approach makes various graph optimisation such as layout rearrangement and constant folding possible, it is difficult to model dynamic graphs [78]. For variable length data in a recurrent neural network, one can append zeros to every sequence so that they have the same length. For a model that creates a different structure of network for each data sample, using static graph is even more improbable. It also makes debugging very difficult because graph execution is a closed-box in which intermediate values are not accessible.

As opposed to early TensorFlow, PyTorch adopts the idea of *'define-by-run'* [76] or *'eager'* where dataflow graphs are constructed on-the-fly during computation at runtime. The same linear example written in PyTorch is shown below:

```
import torch

# creates a tensor with data
x = torch.tensor([1,2,3,4,5,6])
# define a parameter by setting 'requires_grad=True'
W = torch.tensor([2.], requires_grad=True)
b = torch.tensor([1.], requires_grad=True)
# creates a graph for 'y_pred = x * W + b'
y_pred = torch.add(torch.mul(x, W), b)

print(y_pred) # => [3,  5,  7, 9, 11, 13]
```

`torch.tensor` creates a tensor node, while `torch.add` and `torch.mul` create the function node for addition and multiplication respectively. For `W` and `b`, `requires_grad` is set to `True` to denote that they are trainable model parameters. This style of graph construction is arguably more natural than 'define-and-run' as it is how codes are commonly written. In order to train the model, we continue with the training loop:

```
for i in range(100):
  # training data
  x = torch.tensor(x_batch)
  y = torch.tensor(y_batch)

  # creates the graph
  y_pred = torch.add(torch.mul(x, W), b)
  # creates the graph of the loss function
  loss = torch.sum((y - y_pred) ** 2)
  loss.backward()  # back propagation

  with torch.no_grad():
    # manually updates the parameters
    W -= learning_rate * W.grad
    b -= learning_rate * b.grad
```

In 'define-by-run', graphs are constructed incrementally whenever operations are evaluated. When an operation is being evaluated, all the inputs and intermediate values are kept in order for the `loss.backward()` to perform back propagation and to calculate the gradients. Once `loss.backward()` is finished, the graph will be discarded, meaning that a new graph is created every time for each iteration. In this way, dynamic models are easy to implement because the graph can be modified between iterations. However, if the model is actually static, then the reconstruction of the graph would seem unnecessary. Also, because of its dynamic nature, many of the graph optimisation methods are not applicable.

Objective: to design a language where the graph models are dynamic but efficient.

1.2.2 Symmetric optimisation

In many of the frameworks, such as Torch, TensorFlow and PyTorch, upon creating a model, parameters will be automatically added to a global collection. The optimiser will then use this collection to calculate the gradients and to update the parameters. An example written in Torch is shown in Fig. 1.2.

```
require 'nn';
-- loading the training set
trainset = torch.load('src')
-- creates a sequential network
local net = nn.Sequential()
-- creates a fully connected layer with 1 input and 1 output
-- note that a:method(param) is the same as a.method(a, param)
net:add(nn.Linear(1,1))
-- get parameters
local weights, grad_weights = net:getParameters()
-- creates the loss function
local criterion = nn.MSECriterion()

for i = 100 do
  -- data batches
  local input = x_batch;
  local output = y_batch;
  -- feed it to the neural network and the criterion
  local pred = net:forward(input)
  local loss = criterion:forward(pred, output)
  -- train over this example in 3 steps
  -- (1) zero the accumulation of the gradients
  grad_weights:zero()
  -- (2) accumulate gradients
  local dpred = criterion:backward(pred, output))
  local dx = net:backward(input, dpred)
  -- (3) update parameters with a 0.01 learning rate
  weights:add(0.01, grad_weights)
end
```

Figure 1.2: Linear regression, example

The operation `getParameters` (or `get_collection` used by the TensorFlow API) will return an array of parameters that can be used in a generic optimisation algorithm such as gradient descent in order to minimise the `loss` function. Gradients are then calculated and added to the parameters by using `weights:add(0.01, grad_weights)`. Note that for simple models, it is possible to place the parameters in a vector by hand. However, in most cases models are extremely complex and consist of many sub-models (e.g. a model of a power plant).

Since the collection of parameters is often in the form of arrays or 'tensors', it will have the typical issue of dimensionality. When we try to reconstruct or modify the parameters, there

is nothing stopping us from using an array of the wrong size as an argument or accessing the wrong index, causing a runtime error which is *unsafe*. This, one could argue, is the price we need to pay if we are going to use array programming, even in a functional language.

In addition, we cannot expect the programmer to understand the order in which parameters will be put into the collection. For a linear regression model $f(x) = ax + b$ will the a be the first in the array, or the b? Which original parameter does the nth element of a vector of model parameters corresponds to? As such, it is essential to have a general approach that collects all parameters in a vector, then uses a generic optimisation algorithm which is parameterised both by the parameters themselves and by the model.

In this thesis, we shall also address this problem and offer a safe alternative. First, we use generative typing so that each time when parameters of a model are collected they inhabit a new type. This new type is then used by the optimisation algorithm to ensure that the said model can only be updated through the collected parameters. Moreover, this type is 'opaque' and the programmer has no access to the individual elements directly or indirectly (e.g. via the basis of the vector). The type is packaged together with a set of *symmetric* operations, i.e. invariant over any permutation of the basis. This means that the programmer needs to express the optimisation algorithm using these symmetric operations. Perhaps not surprisingly, virtually all optimisation algorithms we considered, numeric (e.g. gradient descent) or combinatorial (e.g. simulated annealing) can be expressed using symmetric operations only.

Objective: to offer a safe alternative to direct array manipulation for parameters optimisation.

1.2.3 A unified semantics

Some languages such as Lustre [22], Lucid Synchrone [20] and most functional reactive programming languages have a semantic model that can be expressed denotationally through

systems of recursive and co-recursive equations. In these cases, the interesting issue is efficiency, particularly how memory should be utilised. A semantically interesting question occurs in the case of languages which provide primitives to manipulate dataflow graphs imperatively. That is, to devise a semantics that unifies dataflow graphs and formal languages, and to prove certain properties such as type soundness and observable equivalences. This is the case of Tensorflow [1], self-adjusting computation [2] (SAC) and most graphical dataflow languages.

Self-adjusting computation [2] is a programming paradigm intended for incremental computation. Its purpose is to minimise re-computation assuming only small changes are made to the input. Although it is not considered as a dataflow language, it is also founded on dataflow graphs programming and the propagation of changes. Consider in the following code written in OCaml:

```
let x = SAC.create 1
let y = SAC.create 2
let m  = SAC.map2 (SAC.watch x) (SAC.watch y) (+)
let mo = SAC.observe m
let _ =
  SAC.stabilize (); print (SAC.value mo);
  SAC.set x 3; SAC.stabilize (); print (SAC.value mo)
```

where `SAC.create` creates an input node, `SAC.watch` creates the edge of input nodes, `SAC.map2` creates an addition node with `x` and `y` as its input, and `SAC.observe m` observes the changes made to `m`. When `SAC.stabilize()` is called, the parts that need to be recomputed are evaluated. `SAC.set` changes the value of an input cell; in this case we use it to change the value of `x`. When `SAC.stabilize()` is called again, changes are propagated through the graph. The program should print `3` then `5`. Because `SAC.stablilze()` propagates changes in one-go, the graph needs to be acyclic to prevent propagation going into an infinite loop. However, acyclic graphs are important for modelling feedbacks in a system, therefore in our language the propagation is done step-by-step, and in each step, a node can only be updated at most once. The semantics model is similar to having a logical clock in synchronous lan-

guages. This will be discussed in more details in chapter 3 when we describe the semantics of the language.

Having the ability to create and to manipulate dataflow graphs has proved to be very convenient. However, circular dependencies are not allowed in SAC as the `stabilize()` function will not terminate. As for TensorFlow, although it is possible to create cycles using `tf.while_loop`, they could be unsafe in the sense that the evaluation of dataflow graphs might not terminate. For example, in the following code

```
1 import tensorflow as tf
2
3 i = tf.constant(0)
4 c = lambda i: tf.less(i, 10)
5 b = lambda i: (tf.multiply(i, 1), )
6 r = tf.while_loop(c, b, [i])
7
8 sess = tf.Session()
9 sess.run(r)
```

`tf.while_loop` receives three arguments: `c` is a callable returning a boolean indicating the condition; `b` is a callable denoting the loop body and `[i]` denotes the loop variables. During graph execution, the program will not terminate as the loop variable `i` is always less than `10`. As opposed to that, one of our major goals is to ensure that our language will always terminate during graph execution. In order to allow *safe* feedback loop in the graph, we adopted a synchronous mode of change propagation. The detailed semantics will be discussed in chapter 3 and chapter 5.

Another important property of a dataflow language is whether graphs are constructed by implicit or explicit lifting [10]. In some languages such as Fran [35], functions and operators need to be explicitly lifted to work on data streams by using a family of $lift_n$ operation:

$$lift_n : (\tau_1 \to \cdots \to \tau_n \to \sigma) \to \tau_1\ stream \to \cdots \to \tau_n\ steam \to \sigma\ stream$$

where $\tau\ stream$ is the datatype for a stream of τ. For instance, one needs to write the following code in Fran to add the value `1` to a steam `s`:

```
sum = (lift2 +) s (lift0 1)
```

On the other hand, some other languages, e.g. FrTime [29] and FlapJax [60], support implicit lifting either by inserting code at compile time or by overloading operators. For example, in FrTime the same program will be rewritten as simply

```
sum = s + 1
```

which will be transformed in compile time to

```
sum = if (or (stream? s) (stream? 1))
          (make-stream (λ() (+ (current-value s) (current-value 1))) s 1)
          (+ s 1)
```

where `make-stream` creates a stream and `current-value` returns the current value of a stream. This code is valid because FrTime is built on top of Scheme, which is dynamically typed.

Our language also addresses this issue and aims to provide implicit lifting in a statically typed setting. The difference between a normal value and a stream is transparent to the programmer, and it is left to semantics to deal with. This will be discussed in details in chapter 3 when we describe the semantics.

Objective: to give a unified formalism that harmonises the semantics of the PCF-like language with that of dataflow.

We choose to express the semantics of our language in an unconventional but convenient diagrammatic semantics inspired by the Dynamic Geometry of Interaction Machine [64, 63] (DGoIM). The semantics is basically a graph-rewriting abstract machine in which the execution of a program is given by a token traversing the graphical representation of the program while collecting information and guiding the graph rewriting. It has the desirable features to model both higher-order computation and dataflow in a convenient unified setting. As

an extra bonus, it can also reason about the costs of computation. As a matter of fact, it was proved to be optimal á la Accattoli [3] for various reduction strategies for the lambda calculus [64]. It has been recently used in defining the semantics of the preliminary language of TSD, Idealised TensorFlow [62] (ITF). A generalised version of the DGoIM has also been used to reason about observational equivalence [39]. Therefore, because of its graphical nature, a DGoIM-style semantics seems to be a natural fit for designing dataflow languages in order to give a unified semantics and to prove that the resulting language is in-principle efficient.

Background of Geometry of Interaction machines Geometry of Interaction [41] (GoI) was introduced by Girard to give dynamic interpretation to proof nets in linear logic [42]. It showed that cut elimination can be modelled by an execution formula acting on the proof. Since cut elimination corresponds to beta-reduction in lambda calculus, GoI also shed lights on giving semantics to functional languages which leads to token passing abstract machines, pioneered by Danos and Regnier [31], and Mackie [57].

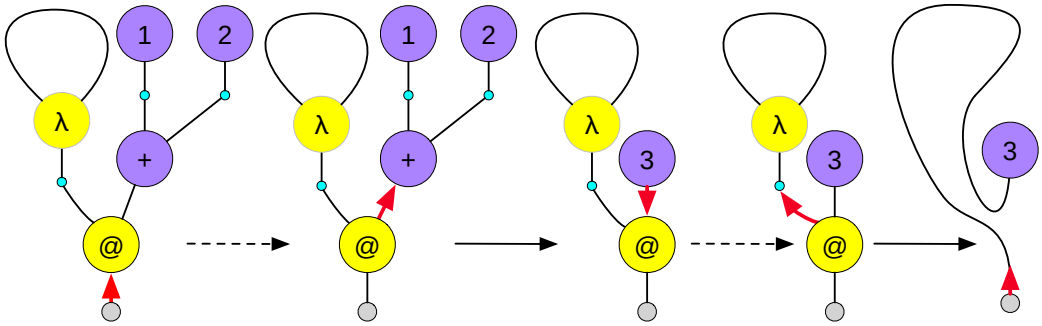
By translating a λ-term into a proof net, a *token passing* GoI machine evaluates a term by passing a data-carrying token which follows a specific path in the net. Throughout the evaluation, the graph remains static, meaning that programs can be executed in constant space. This style of abstract machine has a good track record in leading to compilation schemes for non-standard platforms such as Geometry of Synthesis [38] (GoS) for high-level hardware design; compilation of recursive functions into digital circuits [40]; distributed computing [36]; and compilation with tight memory constraints [72, 54].

Token passing machine implements call-by-name naturally, as the argument sub-graph is traversed by the token every time when needed. However, implementing call-by-need or call-by-value using a static graph is not as straight-forward. Hoshino et al. [48] created a GoI-machine that simulates call-by-value with algebraic effects. While the machine satisfies the equational theory of call-by-value, it does not have the same cost model. In light of this,

Muroya and Ghica [63] created the Dynamic Geometry of Interaction machine by interleaving graph rewriting with token passing. By interleaving these transitions in different ways, all common reduction strategies can be implemented while satisfying their respective equational theory and cost model [64]. In fact, it was proved to be optimal á la Accattoli [3]. As an abstract machine, the DGoI-style semantics allows reasoning about time and space costs of a program or language. But unlike conventional abstract machines, the DGoI approach can be used to reason about type-safety [62] and observational equivalence [39].

Dynamic Geometry of Interaction Machine The DGoIM works similarly to a conventional reduction semantics for a language with explicit substitution, except that terms are represented as graphs. Term constructors (abstraction, application) are nodes in the graph and variables are edges. Reduction is accomplished by the strategic reduction of the graph using a roving *token* (i.e. a selected edge in the graph plus some global information). By 'strategic' we mean that the trajectory of the token is determined by local rules, and that (simple) rewriting only happens at the token. In a conventional semantics, a subtle and often complex aspect of the reduction bureaucracy is the identification and isolation of a subterm called redex from the context. In this abstract machine, the token and its data serve the same purpose. Whatever constructor the token is pointing at is the redex. Narrowing the redex is achieved simply by moving the token along one of its edges.

For instance, the sequence below shows a right-to-left call-by-value evaluation in conventional semantics. Redex candidates are underlined. The transition $\dashrightarrow$ indicates a change of focus, and the arrow $\rightarrow$ a reduction applied to the redex: $\underline{(\lambda x.x)\ (1+2)} \dashrightarrow (\lambda x.x)\ \underline{(1+2)} \rightarrow (\lambda x.x)\ \underline{3} \dashrightarrow \underline{(\lambda x.x)\ 3} \rightarrow 3$.



$\underline{(\lambda x.x)\ (1+2)} \dashrightarrow (\lambda x.x)\ \underline{(1+2)} \to (\lambda x.x)\ \underline{3} \dashrightarrow \underline{(\lambda x.x)\ 3} \to 3.$

Figure 1.3: Evaluation of $(\lambda x.x)\ (1+2)$

The DGoIM represents the term as a graph with nodes λ, @, +, 1, 2 which closely mimics the abstract syntax graph, except that bound variables loop back to their binder rather than receiving a name (x is the name of the bound variable below). The *token* is represented by the (red) arrow on the edges. The change of focus in the term rewriting semantics is replicated by the token motion in the graph shown in Fig. 1.3.

In this thesis, we will build on top of DGoIM, a semantics for TSD by modifying machine states and adding appropriate nodes and transition rules.

1.3 Contributions

Language design In summary, we propose a programming language for systems modelling and optimisation, *transparent synchronous dataflow* (TSD), which is:

1. *imperative* - dataflow graphs are explicitly manipulated using commands;
2. *functional* - the language includes (call-by-value) PCF [69] as a subset;
3. *simply-typed* - the type system is essentially that of PCF;
4. *modal* - dataflow graphs are constructed and can be executed separately;
5. *eager* - dataflow graphs are constructed on-the-fly by computation at runtime;
6. *parallel* - dataflow graphs can be executed in parallel;
7. *dynamic* - dataflow graphs can be changed during execution;

8. *transparent* - the distinction between a graph and a value is transparent to programmers;
9. *synchronous* - the change propagation is safe with feedback loops; and
10. *symmetric* - the result of a program is invariant to permutations of parameters under certain conditions.

Theoretical results For such a programming language, establishing safety and memory efficiency are both challenging and non-trivial problems. Our theoretical contributions are:

1. *specification* - we give a precise mathematical definition of the language using an *abstract machine*, harmonising the semantics of the PCF-like language with that of dataflow;
2. *determinism* - we show that the parallel evaluation of the dataflow graph produces deterministic results;
3. *safety* - we show that for any well-typed program, the abstract machine executes without reaching illegal states;
4. *termination* - we show that the execution of any recursion-free program with dataflow terminates by reaching the final state of the abstract machine;
5. *symmetry* - we show that the result of any program (under certain conditions) is independent of how parameters are collected;
6. *equivalences* - we show that beta-equivalence for usual call-by-value languages holds in our language; and
7. *efficiency* - we show that no operation is 'expensive' in terms of time and memory utilisation.

Methodological results We have also built the following experimental implementations to show that the language is expressive and efficient:

1. a native compiler for DGoIM implementing lambda calculus with naturals; and
2. an OCaml library together with PPX extensions for the TSD language.

1.4 Thesis outline

The rest of the thesis is organised as follows. Chapter 2 discusses the related work in systems modelling languages and its limitations. Chapter 3 describes informally the syntax and semantics of *transparent synchronous dataflow* (TSD). Chapter 4 gives several typical examples in system modelling written in TSD. Chapter 5 presents the formal calculus of TSD. Chapter 6 shows the theoretical results obtained. Chapter 7 walks through various experimental implementations. Chapter 8 concerns with the evaluation of this thesis. Finally, Chapter 9 concludes the thesis and discusses any possible future work.

Chapter 2

Related Work

There has been much research on languages for systems modelling and optimisation. This chapter describes previous work and analyses its limitations. Section 2.1 concerns with various deep learning frameworks, and section 2.2 focuses on reactive programming languages.

2.1 Deep learning frameworks

There have been many frameworks developed for the purpose of deep learning [65, 46]. In some of these frameworks, dataflow graphs are centric to their design. There are two main approaches on how dataflow graphs are constructed: *'define-and-run'* vs *'define-by-run'*. The former approach first creates a dataflow graph and then executes it by pushing data into it. The latter however constructs the graph while computing with data on-the-fly. The two approaches also influence the way automatic differentiation is implemented. In this section, we will review deep learning frameworks based on this categorisation. A summary of the classification of these frameworks is shown in Fig. 2.1. Notice that we will not be discussing too much about other features such as interaction with GPU or distributed learning, as they are not our major concerns at the moment.

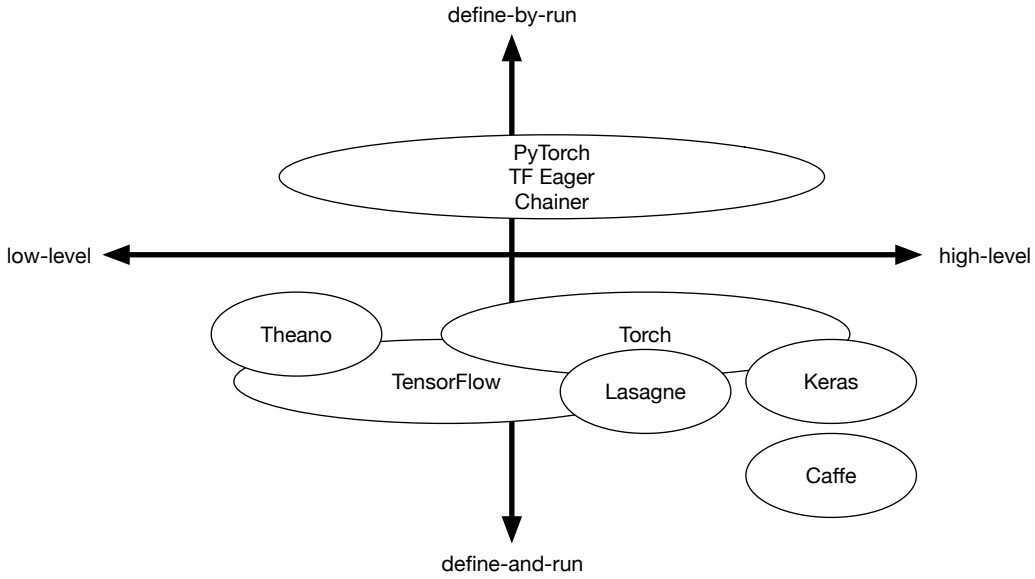


Figure 2.1: Various deep learning frameworks

2.1.1 Define-and-run

Caffe [51] is a C++ and CUDA library which also has interfaces for Python and MATLAB. It is intended to provide an easy way to implement convolution neural networks. In Caffe, graphs are defined in protocol buffers. A simplified MNIST classification model extracted from the official tutorial [18] is shown in Fig. 2.2. Data flows from bottom to top. Firstly, the data layer is defined. It has name 'mnist', type 'Data' and its dataset is given by the LMBD data source with a batch size of 64. It produces two blobs, namely 'data' and 'label', where 'data' is the input values and 'label' is the classification output. Secondly, the convolution layer is defined using 'data' as its inputs. It produces the blob 'conv1'. Finally, the loss function is created in the softmax layer, with 'conv1' and 'label' being its inputs. In order to train the model, we have to define and execute a solver written in another protocol buffer.

Caffe is easy to use because the model and the training languages are declarative. It does not require the user to write any procedural code. It has a clear structure and is human-readable. Since models are defined in protocol buffers, they can be adapted to different execution engines. However, Caffe is not as flexible as other frameworks, such as Theano

```
layer {
  name: "mnist"
  type: "Data"
  transform_param {
    scale: 0.00390625
  }
  data_param {
    // data source
    source: "mnist_train_lmdb"
    backend: LMDB
    batch_size: 64
  }
  top: "data"
  top: "label"
}
```

(a) Data layer

```
layer {
  name: "conv1"
  type: "Convolution"
  // learning rate for weight
  param { lr_mult: 1 }
  // learning rate for bias
  param { lr_mult: 2 }
  convolution_param {
    num_output: 20
    kernel_size: 5
    // convolution stride
    stride: 1
    weight_filler {
      // initialises weights with
      // a randomising algorithm
      type: "xavier"
    }
    bias_filler {
      // initialises bias as constant
      type: "constant"
    }
  }
  bottom: "data"
  top: "conv1"
}
```

(b) Convolution layer

```
layer {
  name: "loss"
  type: "SoftmaxWithLoss"
  bottom: "conv1"
  bottom: "label"
}
```

(c) Softmax layer

Figure 2.2: Model in Caffe

or TensorFlow, as it does not support fine granularity of network layer construction. It also depends highly on the coupling between protocol buffers and their implementations. Custom layers are implemented in C++ by extending the network layer module and overriding some of the methods, such as `Forward()` and `Backward()`. Its name and type are then exported and used in protocol buffers. This means that custom layers have to be maintained in both levels. Furthermore, this style of model construction is inconvenient for creating a large network because the layers are not reusable nor compositional. For example, the ResNet-152[1] is 6775 lines long.

```
require 'nn';
-- loading the training set
trainset = torch.load('src')
-- creates a seqential network
local net = nn.Sequential()
-- creates a fully connected layer with 1 input and 1 output
-- note that a:methond(param) is the same as a.method(a, param)
net:add(nn.Linear(1,1))
-- get parameters
local weights, grad_weights = net:getParameters()
-- creates the loss function
local criterion = nn.MSECriterion()

for i = 100 do
  local input = x_batch; -- data batches
  local output = y_bactch;
  -- feed it to the neural network and the criterion
  local pred = net:forward(input)
  local loss = criterion:forward(pred, output)
  -- train over this example in 3 steps
  -- (1) zero the accumulation of the gradients
  grad_weights:zero()
  -- (2) accumulate gradients
  local dpred = criterion:backward(pred, output))
  local dx = net:backward(input, dpred)
  -- (3) update parameters with a 0.01 learning rate
  weights:add(0.01, grad_weights)
end
```

Figure 2.3: Linear regression in Torch

[1] https://github.com/KaimingHe/deep-residual-networks/blob/master/prototxt/ResNet-152-deploy.prototxt

Torch [28] is a scientific computing framework based on Lua that also has an API written in C. It supports various libraries of machine learning algorithms, including deep learning. It also supports parallel computation through its CUDA implementation. It is also the basis of the development of PyTorch, which will be discussed later. A linear regression example extracted from online tutorials [77, 8] is shown in Fig. 2.3. Unlike Caffe, models in Torch are created using the same language that trains it. Torch is flexible and readable, it provides low-level to high-level model construction and training. For example, the linear regression example can be rewritten as follows:

```
trainset = torch.load('src')
net = nn.Sequential()
net:add(nn.Linear(1,1))
criterion = nn.MSECriterion() -- creates an optimiser
trainer = nn.StochasticGradient(net, criterion)
trainer.learningRate = 0.01
trainer.maxIteration = 100 -- just do 100 epochs of training.
trainer:train(trainset)
```

This allows for fast prototyping of neural networks for experimental purposes. On the other hand, it provides the `nngraph` package allowing programmers to create dataflow graphs arbitrarily as shown in Fig. 2.4.

```lua
require 'torch'
require 'nn'
require 'nngraph'

-- input nodes
local x = nn.Identity()()
local y = nn.Identity()()
local z = nn.Identity()()
-- a = x + y
local a = nn.CAddTable()({x, y})
-- b = a ⊙ z (point-wise multiplication)
local b = nn.CMulTable()({a, z})
-- c = a + b
local c = nn.CAddTable()({a, b})
-- creates the graph module
local mod = nn.gModule({x, y, z}, {c})

-- defines input data
local x_batch = torch.randn(4, 5)
local y_batch = torch.randn(4, 5)
local z_batch = torch.randn(4, 5)
-- executes the graph
local c_obv = mod:forward({x_batch, y_batch, z_batch})
```

Figure 2.4: Dataflow graphs in Torch

However, there is no way to define model parameters directly in the graph. Instead, programmers create a layer using the `nn` module and parameters are automatically created. This gives the programmer less control over designing the structure of the model. Moreover, graphs are constructed in meta-programming style which one could argue to be awkward. This style is also shared by other define-and-run frameworks, such as Theano and TensorFlow, which are going to be discussed soon in this section.

```
import numpy
import theano
import theano.tensor as T

# model parameters
w = theano.shared(rng.randn(),name ='w')
b = theano.shared(rng.randn(),name ='b')

# inputs and outputs placeholders
x = T.vector('x')
y = T.vector('y')

# linear model
y_pred = T.dot(x,w) + b
# mean squared error loss function
loss = T.sum(T.pow(y_pred-y,2))/(2*num_samples)

# create the gradient graphs
gradw = T.grad(loss,w)
gradb = T.grad(loss,b)
train = theano.function([x,y],loss,updates=[(w,w-lr*gradw),(b,b-lr*gradb)])

# training data
X = x_batch
Y = y_batch
for i in range(100):
  finalloss = train(X,Y)
```

Figure 2.5: Linear regression in Theano

Theano [6] is a Python-based library for scientific computing that has been developed into a deep learning library. Its development started since 2007 and at the time it was one of the few open-source deep learning frameworks available. A linear regression example extracted from an online tutorial [71] is shown in Fig. 2.5. It is a low-level language that gives programmers the flexibility of experimenting with unconventional models. Graphs are static and thus a lot of graph optimisations can be applied. However, it only provides a pre-defined gradient descent algorithm and programmers cannot create custom optimisers. Also, because of the static graph nature, it is generally not suitable for constructing dynamic models. This style of graph construction also makes debugging very difficult, as intermediate values of the graph are not accessible and thus cannot be observed during execution.

Theano is also the inspiration and the basis for some higher-level deep learning frameworks such as Keras [26], Lasagne [32] and Blocks [59].

```python
import tensorflow as tf
import numpy as np

# input nodes
x = tf.placeholder(tf.float32)
y = tf.placeholder(tf.float32)

# model parameters
w = tf.Variable(np.random.normal())
b = tf.Variable(np.random.normal())

# predicted output
y_pred = tf.add(tf.multiply(w, x), b)
# mean squared error loss
loss = tf.reduce_mean(tf.square(y_pred - y))

# gradient descent optimiser
optimizer = tf.train.GradientDescentOptimizer(0.01)
# the gradients graph
train = optimizer.minimize(loss)

# initialiser for model paramters
init = tf.global_variables_initializer()

sess = tf.Session()
# initialise model parameters
sess.run(init)
# training data
feed_dict = {x: x_batch, y: y_batch}
for i in range(100):
  sess.run(train, feed_dict)
```

Figure 2.6: Linear regression in early TensorFlow

TensorFlow [1] is a deep learning framework built on top of Python. In early TensorFlow, models are declared through meta-programming which is similar to programming an abstract syntax tree. These models are represented by dataflow graphs that are run via an execution engine. A full example of linear regression extracted from an online tutorial[2] is shown in Fig. 2.6. The first 20 lines construct the linear model where `x` and `y` are the input placeholders

[2] https://github.com/tensorflow/docs/blob/master/site/en/r1/guide/low_level_intro.md

in which the training data will be fed into, `w` and `b` are model parameters, `y_pred` is the actual output, `loss` being the mean squared error loss function, `optimizer` is the default gradient descent optimiser and `train` is the graph that minimise the loss function. After constructing the graph, one needs to define a `session` object in order to run the model with some inputs or to train it with training data.

This meta-programming style of model construction can be awkward as it hinders the use of the full functionality of its host language, such as control flows. For example, the following code cannot be compiled:

```
x = tf.placeholder(tf.float32)

def threshold(x, y):
  if x > y:
    return 1.
  else:
    return 0.

m = threshold(x, 10)
```

Instead conditionals and iterations have to be built into the graph structure by using `tf.cond` and `tf.while_loop` respectively. For instance, in order to create a threshold one needs to use `tf.cond` as follows:

```
m = tf.cond(x > y, lambda: 1., lambda: 0.)
```

In addition, graphs in early TensorFlow are static in the sense that there is no way to reconfigure it in runtime. This makes it very difficult to model dynamic networks. For example, TensorFlow deals with neural networks with variable length inputs by padding all the input data such that they all have the same size as the maximum input data, and masking the appropriate gradient. The TensorFlow Fold package [56] also tries to rectify this problem by introducing dynamic batching. However, even then it is difficult to model recursive neural networks [73] such as TreeNets where the tree structure can be different for each sample input. However, this static nature of early TensorFlow graphs is beneficial for performance

since a lot of optimisation can be done on them. It is also very suitable for interacting with automatic differentiation, as the generated graph is guaranteed to be 'straight-line code'.

2.1.2 Define-by-run

```python
import torch

# model parameters
w = torch.randn(1, 1, requires_grad=True)
b = torch.randn(1, 1, requires_grad=True)

# training loop
for i in range(100):
  # training data
  x = torch.randn(32, 1, ...)
  y = torch.randn(32, 1, ...)
  # model and loss function
  y_pred = torch.add(torch.mul(x, W), b)
  loss = torch.sum((y - y_pred) ** 2)
  # back propagation
  loss.backward()

  with torch.no_grad():
    # updates model parameters
    w -= learning_rate * w.grad
    b -= learning_rate * b.grad
```

Figure 2.7: Linear regression in PyTorch

PyTorch [68] is a deep learning framework that is also built on top of Python. Instead of static graphs, PyTorch advocates the use of dynamic graphs to represent training models, an idea that was first introduced by Chainer [76]. An example of linear regression extracted from an online tutorial[3] is shown in Fig. 2.7. The tensors `x` and `y` hold the training data and are the inputs to the dataflow graph. `w` and `b` are the model parameters. `torch.add` and `torch.mul` are operations that create function nodes. In PyTorch, the graph is not created until the operations are evaluated at runtime; and a new graph is created every time for each iteration in the `for`-loop. Next `loss.backward()` performs back propagation to calculate the gradients of every parameter. After the function was called, the graph will be freed and

[3] `https://pytorch.org/tutorials/beginner/pytorch_with_examples.html`

parameters can be updated using `w.grad` which returns the calculated gradients of `w`.

Since the graph is not constructed until the operations are evaluated, it can be modified in each iteration. Therefore, PyTorch is very suitable for creating dynamic models. However, if the graph is actually static, the cycle of destroying and recreating the same graph structure would be wasteful. Also, because of this, much of the graph optimisations used by Theano and TensorFlow cannot apply. On the other hand, PyTorch has better integration to its host language than TensorFlow as most of the control flows provided by Python can be used to construct graph models.

```
@static_code
def static_forward(self, inputs, outputs):
  # this function will be called for each iteration in the training loop
  x = inputs[0]
  y = outputs[0]
  # Read from x, write results into y in-place.
  y[:] = 5.0*x

# this is the function being called when an object is instantiatied
# since it is not wrapped by @static_code, it will only be called once when
# _supports_static_optimizations is set to True
def forward(self, inputs):
  # Allocate an empty output array.
  y = xp.empty(y_shape).astype(x.dtype)

  # Call static function
  # (it will get called every iteration from optimized schedule)
  self.static_forward(inputs=[x], outputs=[y])
  return y,
```

Figure 2.8: Static subgraph in Chainer

Chainer [76] is a Python-based deep learning framework and the first to introduce the concept of 'define-by-run'. In order to rectify the performance penalty from having to create and discard dataflow graphs repetitively, it also includes *static subgraph optimisations* that allow programmers to declare some part of a graph static [23]. For example, in order to construct a function node in the dataflow graph, one can create a class extending the `FunctionNode` class and override the function `forward` as shown in Fig. 2.8 extracted from

the official tutorial [23]. By setting the class attribute `_supports_static_optimizations` to `True` and putting the `@static_code` tag, the function `static_forward` will be called in each iteration and `forward` will only be called once to set up the static graph.

```
1  import tensorflow as tf
2
3  # model parameters
4  w = tf.Variable(1.0, name='w')
5  b = tf.Variable(0.0, name='b')
6
7  # training loop
8  for i in range( num_epochs ) :
9    # training data
10   x = ...
11   y = ...
12
13   # creates a tape for back propagation
14   with tf.GradientTape() as tape:
15     # linear model
16     y_pred = tf.add(tf.multiply(w, x), b)
17     # loss function
18     loss = tf.reduce_mean( tf.square( y - y_pred ) )
19     # calculates the gradients
20     [dl_dw, dl_db] = tape.gradient(loss, [w, b])
21     # updates the parameters
22     w.assign_add(learning_rate * dl_dw)
23     b.assign_add(learning_rate * dl_db)
```

Figure 2.9: Linear regression in TF Eager

TensorFlow Eager [4] is an extension to early TensorFlow that evaluates operations immediately. It is created to mimic the dynamic model construction of 'define-by-run'. A linear regression example extracted from the official tutorial [75] is shown in Fig. 2.9. In Line 14, a gradient tape is created to log the intermediate values of the model. When `tape.gradient(...)` is called, it uses these values to perform back propagation and calculate the gradients. This example is very similar to that of PyTorch.

The 'eager' extension makes the programming style more intuitive, since `session` is no longer needed to evaluate a model. It has also made debugging easier because values can simply be returned from operations and could be printed out easily. It also makes possible

for creating dynamic models as it now uses the control flows from the host language instead of graph control flow.

2.1.3 Idealised TensorFlow

Idealised TensorFlow [62] (ITF) is a pure functional language inspired by TensorFlow and developed by Koko Muroya, Dan Ghica and the author. It is a preliminary design of TSD and is now obsolete. It is a toy language that introduces the concept of *graph abstraction* which automatically collects parameters from a model and creates a 'skeleton function' that can be used to reconstruct the model with different parameters. The same linear regression program is shown below:

```
1  (* creates the model as a function with `pc' denoting model parameters *)
2  let m = λx. pc 1 * x + pc 0 in
3  (* defines the loss function *)
4  let loss = λy.λy_exp. (y - y_exp) * (y - y_exp) in
5  (* graph abstraction *)
6  (A_a(f, ps).
7    (* gets new parameters by training *)
8    let newps = train f ps loss data in
9    (* gets an updated model *)
10   let new_m = f newps in
11   ⋮
12 ) m
```

Line 2 creates a model by defining a linear function `m` with parameters denoted by `pc 1` and `pc 0`. Line 4 defines the mean squared error loss function as usual. In Line 6, $\mathcal{A}_a(f, ps)$ is the graph abstraction binder applied to the model $m : \mathbb{F} \to \mathbb{F}$. Graph abstraction creates two bound variables: (1) a vector of collected parameters ps : $Vec\ a$; and (2) a 'skeleton function' f : $Vec\ a \to \mathbb{F} \to \mathbb{F}$ where $ps \equiv [1; 0]$ and $f \equiv \lambda v.\lambda x.\pi_0(v) * x + \pi_1(v)$ with $\pi_i(v)$ being the i-th projection of v. Line 8 trains a new vector of parameters, and Line 10 creates an updated model by applying the skeleton function f to the updated parameters.

This style of programming is very elegant and natural. Model execution and training are simply function applications. It also has a deeper meaning in terms of programmable

induction and abduction [24]. However, this language is far from complete as it lacks common control flows such as conditional and recursion. This makes it very difficult to create large models with plenty of parameters, as they will have to be created one-by-one manually. Even if the language were to be extended with recursion, the problem will not be solved because parameters in ITF act like global references. They are always shared, but not copied. Consider the following code:

```
let m = rec f. λn.
  if n <= 0 then 0 else pc 1 + f (n-1)
in
let y = m 10 in
(A_a(f, ps). ...) m
```

After execution, the graph `y` is only connected to only one parameter `pc 1` because it is not copied. The same happens to graph abstraction, and `ps` is only bound to `[1]`.

More importantly, ITF lacks the concept of change propagation. Although one can create computation graphs in ITF, they do not have any changeable input nodes and thus cannot be considered as dataflow graphs. Therefore, when using ITF to train model parameters, the same model is duplicated many times, which is inefficient.

2.2 Reactive programming languages

Reactive programming is a programming paradigm concerned with time-varying values, i.e. *streams*, and the *propagation of changes* [10]. Its major usage is to program reactive systems. It has also gained popularity among the development of interactive and event-driven applications, for instance interactive user interfaces. This paradigm provides the abstractions to express streams and transparent means to maintain data dependencies in the program. There have been many approaches to reactive programming. For example, *synchronous*, where reactions are governed by one or more clocks, versus *asynchronous*, where the system reacts to an event as soon as it is received. In this section, we will only describe the areas that are mostly related to our work: synchronous dataflow languages and self-adjusting computation.

The latter although originally intended as a paradigm for incremental computation [70] has also found applications in creating user interface and web programming.

2.2.1 Synchronous dataflow languages

Synchronous languages are designed specifically to model, to specify and to implement real-time reactive systems [12]. Lustre [45] and Esterel [13] were among the first synchronous languages created in the 1980s. These languages are designed around the assumption of 'synchrony' where time advances in lockstep in one or more clocks. Under this assumption, the reactions of the system only take a finite amount of time and memory and are atomic while the external environment stays unchanged during the execution of the reactions. In order to model safety-critical systems, the design goal of these languages is to combine synchrony with deterministic concurrency in a simple manner such that formal reasoning is tractable.

Lustre Lustre [45] is a declarative language where dataflow graphs are represented as systems of equations. In such formalism, variables are treated as streams which can only be assigned once. For example, when one writes $x = m + n$, it means that for any time instant k, $x_k = m_k + n_k$. It also provides basic primitives akin to that of Lucid [7] for manipulating streams: `pre` (previous), `->` (initialisation) , `when` (sampling) and `current` (interpolation). Suppose `X` is the stream of natural numbers and B is an alternating boolean stream, these operators will generate streams as follows:

`X`	0	1	2	3	4	5	6	7	...
`pre(X)`		0	1	2	3	4	5	6	...
`a -> X`	a	1	2	3	4	5	6	7	...
`B`	t	f	t	f	t	f	t	f	...
`X when B`	0		2		4		6		...
`current(X when B)`	0	0	2	2	4	4	6	6	...

As shown in the example, the `pre` operator will create an undefined instance and Lustre has built-in initialisation analysis to check if any undefined values are used by other parts of the program. It is also illegal to write `X = X + 1` as it will create an infinite loop. This is also checked by a causality analysis.

Lustre is a subset of Lucid, but extended with clocks. The rules of the clock calculus is defined in [22]. By default, every stream has the basic clock which is a boolean stream with constant `true` value. A flow can have a slower clock by using sampling, for instance, `X when B`. Below is an example of an order-2 FIR filter mentioned in the introduction:

```
node fir2(X: float, b0: float, b1: float, b2: float) returns (Y: float);
let
  /* X(n-1) */
  X1 = pre X
  /* X(n-2) */
  X2 = pre X1
  /* output */
  Y = b0 * X + b1 * X1 + b2 * X2
tel

Out = fir2(X, 1/3, 1/3, 1/3)
```

Below are some more examples of streams generation in Lustre:

```
N = 0 -> pre(N) + 1; /* stream of naturals */
Y = 0 -> pre(Y) + X; /* running sum of X */
F = 1 -> pre(F + (0 -> pre(F))); /* Fibonacci */
```

Dataflow graphs are constructed in Lustre by declaring equations between streams. The resulting code is elegant and succinct. However, Lustre is strictly a first order language and treats streams as first order data. Also, the dataflow graph is static and cannot be reconfigured during execution.

Lucid Synchrone and Zélus Later Lucid Synchrone [21] (LS) brought the functional perspective into Lustre. It is a strongly-typed language that supports higher-order functions and other functional features such as pattern matching. On top of the Lustre primitives, LS also

introduced the `merge` operation which, unlike the if-statement that selects elements from two streams, combines two streams of complementary clocks into a single stream. For example

X	0	1	2	3	4	5	6	7	...
B	t	f	t	f	t	f	t	f	...
Y = X when B	0		2		4		6		...
Z = X when not B		1		3		5		7	...
merge B Y Z	0	1	2	3	4	5	6	7	...

The safety of `merge` is checked by the clock calculus. For instance, the stream `Y` has a clock type of `B` while `Z` has clock type $\overline{\texttt{B}}$. In addition, because of the functional nature it also introduces the concept of 'streams of functions' which can be used to model reconfigurable systems. Therefore, graphs in LS are dynamic as opposed to Lustre where graphs are static. Later versions of LS also include features that create state machines and manipulate signals, which resemble the language constructs in Esterel [27]. Later Zélus [16] was created based on LS extended with features for defining and solving ordinary differential equations.

Esterel and ReactiveML Unlike Lustre which is data-oriented and declarative, Esterel [13] is event-oriented and imperative. Instead of defining the dataflow graph as in Lustre, programs in Esterel define what happens when events are triggered. An Esterel program consists of multiple concurrent processes written in imperative syntax that are executed under a global synchronous clock. At the start of each reaction, a process continues from where it was paused in the last reaction, executes its codes sequentially and lastly either terminates or reaches `pause` which will be resumed in the next reaction. These processes communicate only through global signals, in which a process can `await` to receive an event from as well as `emit` to signals. It is very suitable to model state machines, for instance a typical ABRO state machine can be implemented as follows:

```
1 module ABRO:
2 input A, B, R;
```

```
output O;

loop [ await A || await B ];
  emit O;
each R
end module
```

This state machine emits an output to `O` upon receiving an event from either `A` or `B`, and reset upon `R` is present. This style of programming is very suitable for describing controls in concurrent systems. However, deadlock can be created very easily. Consider the following code written in ReactiveML [58], a language built on top of OCaml combining the primitives of Esterel with functional features:

```
signal x, y

let process a =
  await x in
  emit y

let process b =
  await y in
  emit x

run (a || b)
```

Process `a` will only emit `y` if it receives an event of `x` while process `b` cannot emit `x` unless `y` is triggered. A causality analysis is required to detect these deadlocks.

2.2.2 Self-adjusting computation

Self-adjusting computation [2] (SAC) is a programming paradigm for incremental computation. It concerns with minimising re-computation when the inputs are slightly changed. There are two major ingredients in SAC: 1) *change propagation* and 2) *memoisation*. We are interested in the former.

In SAC, a special syntactic and semantic category of *cells* or *variables* is introduced, with all terms depending on these cells being automatically recomputed on-demand when the cell is changed. The changing of a cell triggers an event, which automatically propagates

along a dataflow graph tracking the dependencies, until it can be observed at the other end. A commonly invoked metaphor is that of the spreadsheet, where changing data in a cell automatically triggers the recalculation of all the dependent cells, their dependants, and so on. Below we give a simple example, in a simplified syntax inspired by a popular SAC library [50].

```
let xv = SAC.create 1
let yv = SAC.create 2
let x  = SAC.watch xv
let y  = SAC.watch yv
let m  = SAC.map2 x y max
let mo = SAC.observe m
let _ =
  SAC.stabilize (); print (SAC.value mo);
  SAC.set xv 3; SAC.stabilize (); print (SAC.value mo)
```

We use the SAC module name to indicate the provenance of functions for creating variables, observing computations, and retrieving values of computations. Note that the "stabilize" operation triggers the automatic re-computations of all the dependent computations. The program above should print first 2 then 3.

One problem of SAC is that it does not allow feedback loops to be constructed because `stabilise()` will diverge. In fact, in the JaneStreet library, loops are detected at runtime and exceptions are thrown.

Chapter 3

Transparent Synchronous Dataflow

Before technical details are provided, let us try to understand the intended semantics of the language via some examples. This informal introduction will be separated into several parts: i) the dataflow core concerning the manipulation of graphs in section 3.1, ii) the extensions for optimising model parameters in section 3.2, iii) polymorphism via syntactical copying in section 3.3 and iv) the abstract machine semantics in section 3.4.

As described in the introduction, there are three main objectives to the design of TSD: 1) to create straight-line codes transparently; 2) to support symmetric optimisation safely; and 3) to have a semantics that unifies computation graph and dataflow graph. We will also describe how they are achieved in this section.

3.1 The dataflow core

The starting language is call-by-value PCF, a typical language combining the simply-typed lambda calculus with arithmetic-logical operations and recursion. To it, we add a new type of *Cells* along with facilities for manipulating them. We also replace the type *Integer* and its operations with an ordered field $(\mathbb{F}, +, -, *, \div, \leq)$ since the main purpose of the language is to deal with numerical optimisation. The distinction between the type of field and type of cell is more subtle than in languages with assignable state: the inputs and outputs of

the dataflow graph are the edges and have type $\mathbb{F}$ whereas stateful nodes in the dataflow graph have type *Cell*. Note that in our graphs, the edges represent the dependency relations between nodes, therefore the input of a node would be represented by its outgoing edge (on top) while the output of its incoming edge (at the bottom). For example, a cell in the graph will have a single incoming edge representing the use of this cell and a single outgoing edge connecting to its dependency computation.

The key new operations used to manipulate dataflow graphs are:

`ref` : $\mathbb{F} \to \mathit{Cell}$ creates a new cell and returns the edge of the cell initialised with a field value;

`deref` : $\mathit{Cell} \to \mathbb{F}$ takes a cell in the dataflow graph and returns its incoming edge (output);

`root` : $\mathit{Cell} \to \mathbb{F}$ takes a cell in the dataflow graph and returns its outgoing edge (input); and

`link` : $\mathit{Cell} \to \mathbb{F} \to \mathit{Unit}$ takes a cell and changes its outgoing edge (input) to the edge given as an argument.

To construct a program, it is convenient to also have more conventional assignment and dereferencing operations for instantly changing and reading the values of cells rather than just manipulating dependencies. In these operations, the type $\mathbb{F}$ is interpreted in the conventional way, as a value. Therefore, the type $\mathbb{F}$ can either denote a normal value or a dataflow graph. This dual treatment of the type of field may seem ambiguous, but it is not, because each field *value* is also an immutable node in the dataflow graph. Taking the value itself or the edge of the node means the same thing.

`assign` : $\mathit{Cell} \to \mathbb{F} \to \mathit{Unit}$ changes the value of a cell instantly; and

`peek` : $\mathbb{F} \to \mathbb{F}$ creates a constant node and returns its incoming edge (output).

Finally, the command `step` : $\mathbb{F}$ activates the computation in the dataflow graph by updating the contents of each cell in parallel, and at most once, if its dependencies have changed. This allows computing in dataflow graphs with cycles, provided that each cycle contains at least one cell. In effect, the model of propagation is synchronous, with the cells serving as synchronisation points. The command will return either 1 or 0 indicating there is or isn't any update. Knowing if there is any update is a useful convenience, as it may allow repeating the

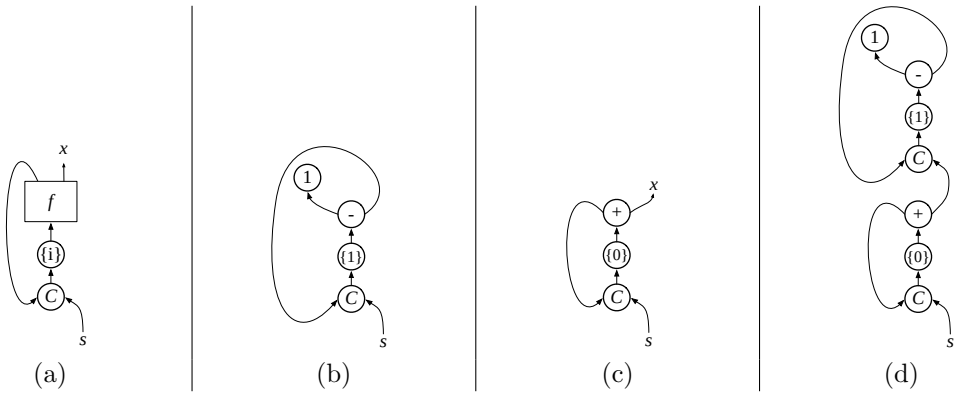


Figure 3.1: Informal example for state machines

propagation step until the graph 'stabilises', i.e. no more updates happen. This is normally possible if and only if the dataflow graph lacks cycles.

$$\texttt{let } \mathit{stabilise} = \texttt{rec } f.\texttt{if step then } f \texttt{ else } 1$$

For the purpose of examples, we consider a standard `let` construct as syntactic sugar. Let us now consider a 'hello world' style example for dataflow computation: state machines. A generic state machine is created in our language by a function (sm) taking as arguments an initial value i, a transition function f, and an input x. Note that the sequencing operation `p;q` is simply defined as a syntactic sugar of `let` $y = p$ `in` q.

$$\texttt{let } sm = \lambda i.\lambda f.\lambda x.\texttt{let } s = \texttt{ref } i \texttt{ in link } s\ (f\ s\ x);\ \texttt{deref } s$$

The function creates a state `s` by initialising it with `i` and linking it back to the transition function $f\ s\ x$. Because of the way i is used as an argument to cell creation, we can think of it as a plain value. The resulting graph will have outgoing edge x, and incoming edge `deref` s. The resulting dataflow will have the general shape given in Fig. 3.1(a). Besides the i-initialised cell-node, it will also include a special 'contraction' node C so that the incoming edge can be used both as output and as cyclic dependency. The function f will be used, during execution, to create a desired dataflow graph.

Let us call the function above sm and consider some simple instantiations, the examples

are also available in the online visualiser[1]:

oscillator is a state machine which alternates between 0 and 1, defined as

$$\texttt{let}\, alt = sm\, 1\, (\lambda s.\lambda i.1 - \texttt{deref}\, s)\, 0$$

Note that the outgoing edge is not used, and 0 is a dummy value. The created dataflow graph is represented informally in Fig.3.1(b), where the sub-graph produced when the particular transition function was supplied now has nodes for subtraction and the constant 1. The cell is initialised to 1. The dummy value 0 is not shown in the figure for convenience, but it does not affect the semantics because it is not reachable by the root of the graph.

sum is a state machine which accumulates the value of its inputs into the state, defined as

$$\texttt{let}\ sum = \lambda x.sm\ 0\ (\lambda s.\lambda i.i + \texttt{deref}\ s)\ x$$

The dataflow graph is sketched out in Fig. 3.1(c).

composite is a state machine which connects the incoming edge of the oscillator to the outgoing edge of the accumulator, defined as

$$\texttt{let}\ a = sum\ alt$$

It can be seen in Fig. 3.1(d).

The code above only constructs the dataflow graph. To execute it, we need to execute the `step` command along with `peek a` to retrieve the output.

$$\texttt{step};\ \texttt{peek}\ a;\ \texttt{step};\ \texttt{peek}\ a$$

After the first `step` the first cell will change its stored value from 1 to 0, and the second one from 0 to 1. Therefore, the first `peek` a will return 1. After another `step` the first cell will further change to 1 and the second remains 1 because it recomputes the addition using the previous value of the first cell (which is 0). Therefore, the next `peek` a also returns 1.

[1] `https://cwtsteven.github.io/TSD-visual?ex=alt`

3.2 Extensions for parameters optimisation

In order to grant the ability to optimise model parameters, we extend the dataflow core with name-indexed types *Vec a* (vectors of $\mathbb{F}$) and *Prm a* (modifiable vectors of $\mathbb{F}$). A member of both types is essentially a vector of $\mathbb{F}$ but the elements of a vector of type *Prm a* can be modified using some operation that will be discussed soon. Their relation is akin to that of $\mathbb{F}$ vs *Cell*.

Consider the following code:

$$\texttt{let } x = \texttt{ref } 1 \texttt{ in let } \mathit{model} = x * \texttt{pc } 1 + \texttt{pc } 0 \texttt{ in } (F(f, ps) \dots) \mathit{model}$$

the model contains two provisional constants 1 and 0 which are created using the `pc` operator, whereas $F(f, ps)$ is the fusion binder. When the fusion binder is applied to the model, all the provisional constants reachable from the model will be collected into $ps : \mathit{Prm}\ a$ which is either $[1; 0]$ or $[0; 1]$ depending on how the graph is traversed. The name a is generated by the fusion binder such that this is the only type that can be used to subsequently update the model. $f : \mathit{Vec}\ a \to \mathbb{F}$ is a function that reconstructs the model upon applying to a vector. We call this function the 'skeleton' function, as it retains the structure of the original model. In this example, a possible outcome of ps and f is:

$$ps : \mathit{Prm}\ a \equiv [1; 2] \qquad f : \mathit{Vec}\ a \to \mathbb{F} \equiv \lambda ps.\ \texttt{deref } x * \pi_0(ps) + \pi_1(ps)$$

where π_i is the projection of the i-element of a collection of parameters.

Similar to the dataflow core, some new operations are provided for manipulating parameters:

`derefp` : $\mathit{Vec}\ a \to \mathit{Vec}\ a$ takes a collection of parameters in the dataflow graph and returns its incoming edge (output);

`assignp` : $\mathit{Vec}\ a \to \mathit{Vec}\ a \to \mathit{Unit}$ changes the value of the parameters instantly;

`peekp` : $\mathit{Vec}\ a \to \mathit{Vec}\ a$ creates a constant node and returns its incoming edge (output).

In addition, we also provide a set of symmetric vector operations, i.e. invariant under permutation:

$\boxplus : \mathit{Vec}\ a \to \mathit{Vec}\ a \to \mathit{Vec}\ a$ is vector addition;

$\boxtimes : \mathbb{F} \to \mathit{Vec}\ a \to \mathit{Vec}\ a \to \mathit{Vec}\ a$ is scalar multiplication;

$\odot : \mathit{Vec}\ a \to \mathit{Vec}\ a \to \mathbb{F}$ is dot product; and

`foldp` : $(\mathit{Vec}\ a \to \mathit{Vec}\ a) \to \mathit{Prm}\ a \to \mathit{Vec}\ a$ applies the function over the base vectors and aggregates the result.

The `foldp` operation allows programmers to apply a function over each dimension of a vector, and then fold over the result. We will show that this operation is used to create the gradient descent optimiser later in section 4.1.

The operation defined as

$$\texttt{foldp}\ g\ ps \equiv \texttt{fold_right}\ (\lambda x.\lambda acc.acc \boxplus g\ x)\ \texttt{basis}\ \vec{\mathbf{0}}$$

where `fold_right:('a -> 'b -> 'b) -> 'a list -> 'b -> 'b` is the *right*-fold of a list, `basis` is the list of base vectors $[b_1^n, \ldots, b_n^n]$ such that n is the length of ps and $\vec{\mathbf{0}}$ is the zero vector. Although ps does not appear in the right-hand side of the definition, `basis` is actually defined by its length. Note that the basis is not accessible in the language.

The use of symmetric operations inhibits the programmer from accessing individual elements in a vector under certain circumstances, which avoids the problems of dimensionality. It is perhaps easy to see that these operations are indeed symmetric, except for `foldp`. In section 6.9, we will show that `foldp` is also symmetric under certain conditions and how optimisation algorithms such as gradient descent can be expressed by it.

3.3 Syntactic name-polymorphism

Since fresh names are generated every time fusion is called, one would expect to be able to write generic functions that can be applied to different names, hence name-polymorphism. The type system is similar to that of Hindley–Milner [30, 47, 61] with types and type schemes but only functions in canonical form, i.e. an abstraction, can be declared polymorphic. The implementation is similar to *function templates* in C++ [19] where the compiler generates

different copies of the polymorphic function with different types. For example, in C++, one could define a generic function that returns the minimum of two values as follows:

```
template <typename T> // 'T' is the type variable
T min (T a, T b) { return a < b? a : b; }

int main() {
  int x = min(10, 20);
  double y = min(30.9, 20.4)
}
```

The compiler will generate codes for two copies of `min`, one with type `int`,

```
int min(int a, int b) { return a < b? a : b}
```

and one with `double`

```
double min(double a, double b) { return a < b? a : b}
```

and points the caller to the corresponding function.

The same technique is employed by TSD to achieve name-polymorphism. A source program written in TSD with polymorphic names is transformed into an intermediate language, *simple-TSD* (simTSD), with only simple types. For every instance of a variable with polymorphic names in TSD, the transformation copies its corresponding code and generates a new variable pointing towards the copy and replaces the instance. For example, consider the following program in TSD:

```
let h = λx. x ⊞ x in
fuse (f,x) from m1 in
fuse (g,y) from m2 in
h x; h y
```

The function h is polymorphic and should have type $\forall a.\, Vec\ a \rightarrow Vec\ a$ which is then applied to $x : Vec\ b$ and $y : Vec\ c$. This program will be transformed into simTSD as follows:

```
let h1 = λx. x ⊞ x in
let h2 = λx. x ⊞ x in
fuse (f,x) from m1 in
```

```
4 fuse (g,y) from m2 in
5 h1 x; h2 y
```

Since there is no universal quantifier in the type system; therefore, $h1 : \mathit{Vec}\ b \to \mathit{Vec}\ b$ and $h2 : \mathit{Vec}\ c \to \mathit{Vec}\ c$.

The major restriction is that only functions in canonical form, i.e. $\lambda x.t$, can be considered polymorphic because otherwise effects will be copied and duplicated. For example, if

$$\texttt{let}\ f = \texttt{let}\ z = \texttt{step}\ \texttt{in}\ \lambda x.x \boxplus x$$

is considered polymorphic the `step` command will be executed more than once because the function f is copied. The reason we adopt syntactic polymorphism is that in order to provide a full Hindley-Milner style polymorphism, it would require renaming of the graph at runtime, which introduces complications that are not well-handled by the current formalism of the semantics. Moreover, name-polymorphism is not the main objective of our language, although we believe that it is possible with a more sophisticated abstract machine.

3.4 Abstract machine semantics

The abstract machine that is used to present the semantics is inspired by the Dynamic Geometry of Interaction machine [64] (DGoIM) which was introduced in previous chapter in section 1.2.3. From there, we describe the semantics of TSD informally in this section.

Basics Let us first consider a basic lambda term, with no dataflow. For simple expressions, the computation graph resembles the syntax tree. Constants and operations are nodes and variables are edges. Abstraction is represented as a cycle in the graph. Contraction (a variable used many times) is handled by dedicated contraction nodes. All these features are illustrated by the term[2] $(\lambda x.x + x + 1)\ 2$. The λ-node and @-node correspond to abstraction and application. Other (cyan) elements, C-node and !-node, are linear logic artefacts for

[2] https://cwtsteven.github.io/TSD-visual/?ex=basic

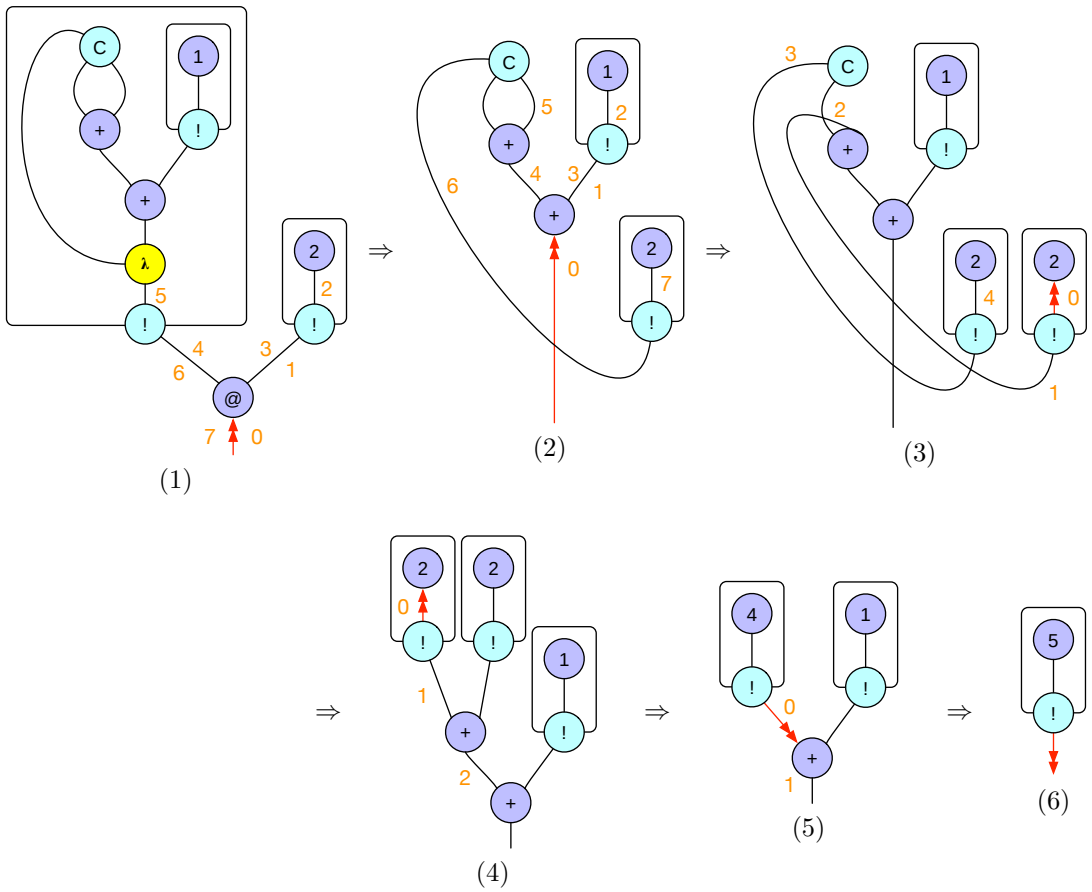


Figure 3.2: Basic example $(\lambda x.x + x + 1)\ 2$

managing the bureaucracy of copying or sharing sub-graphs. Finally, some sub-graphs are identified in 'boxes' which means that they are treated as a whole for the purpose of sharing or copying. The sequence of rewrites is in Fig. 3.2. The token is a double (red) arrow. The (orange) numbers along the edges show the trajectory of the token when no significant rewrites occur. After visiting the argument of the first application, as right-to-left CBV evaluation requires, the first rewrite is equivalent to beta-reduction where the λ-@ pair of nodes is eliminated, joining the edge corresponding to the argument of the function to the argument 2. After a few transitions, the token reaches the contraction node (C), which triggers the copying of the argument 2, thus completing the substitution. The final few steps reduce the arithmetic expression $2 + 2 + 1$ to the constant 5.

Dataflow graph construction Now let us look at how dataflow can be described in DGoIM. A TSD program works in two modes. The first mode is the normal CBV evaluation, which is the mode in which the dataflow graph is created. Or, to be more precise, this is the dataflow sub-graph, which is embedded into the computation graph described above.

The dataflow graph is an elaboration of the basic concept of 'value', which can be in turn thought of as a trivial, degenerate dataflow graph. The most subtle aspect of the dataflow graph is the blending of sharing and copying during its construction. Consider the term[3]

$$(\lambda x.\lambda y.(y + y) + (x + x))\ (\texttt{deref}\ (\texttt{ref}\ 0))\ 2$$

a function applied to both a cell and a conventional constant (Fig 3.3 3.4). The m-node represents the `ref` operation which creates a cell at runtime; while d-node is the `deref` operation. The ?-node is dual to the !-node, in which !-node is the entry of a box and ?-node is the exit. It corresponds to a variable used in the term. The first significant rewrite is replacing the computation graph of the sub-term `ref` 0 by the dataflow sub-graph consisting of a cell ($\{0\}$-node) with initial value 0. After two beta rewrites, the result is a dataflow graph (4), but that is not its final form. The DGoIM will reduce all the sub-terms that can

[3] `https://cwtsteven.github.io/TSD-visual/?ex=dataflow`

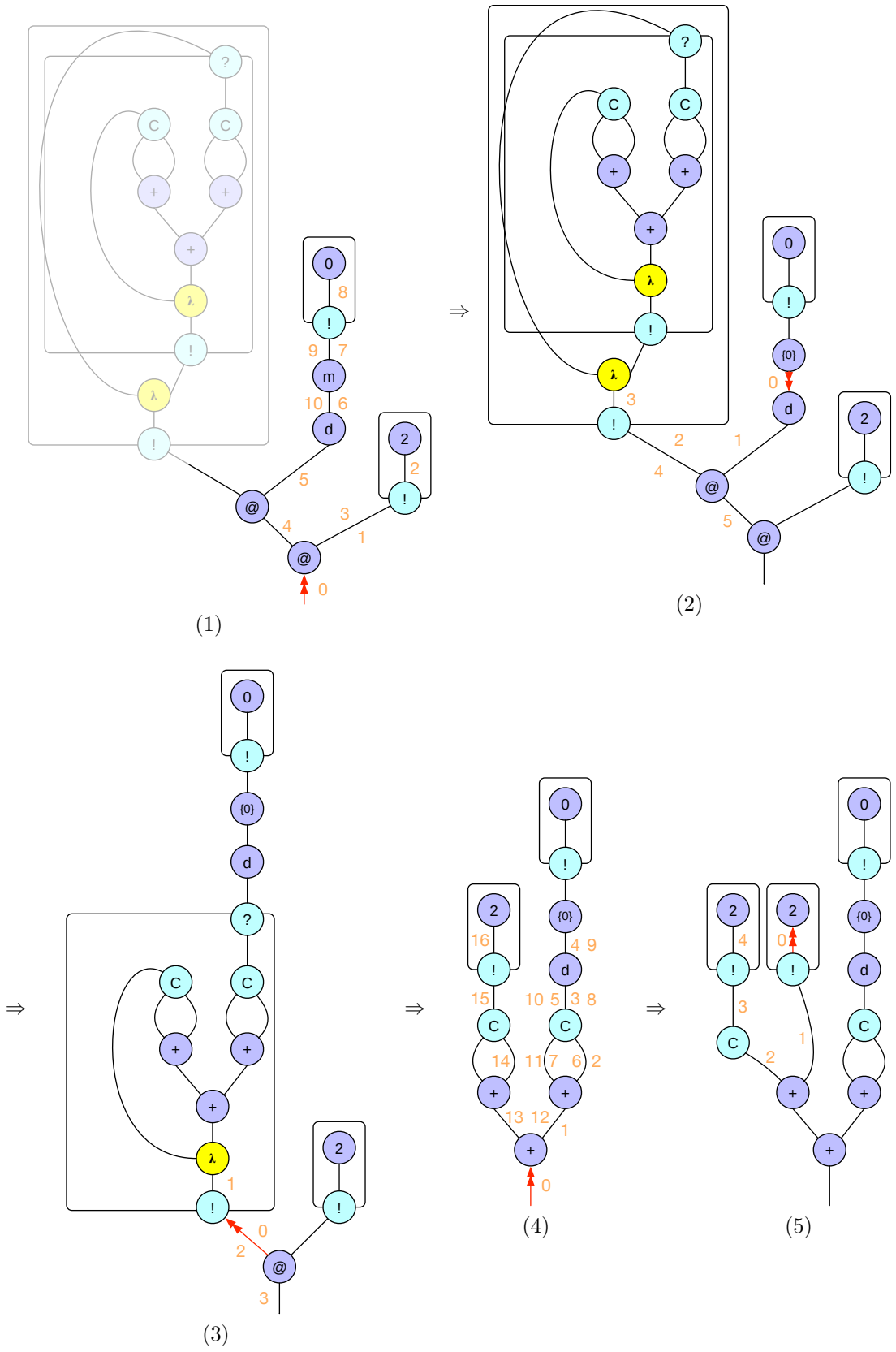


Figure 3.3: Dataflow graph $(\lambda x.\lambda y.(y + y) + (x + x))$ (`deref` (`ref` 0)) 2

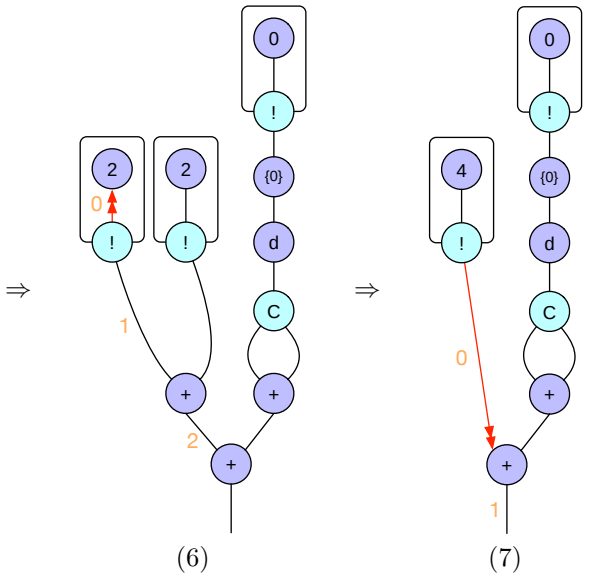


Figure 3.4: Dataflow graph $(\lambda x.\lambda y.(y+y)+(x+x))$ (`deref` (`ref` 0)) 2

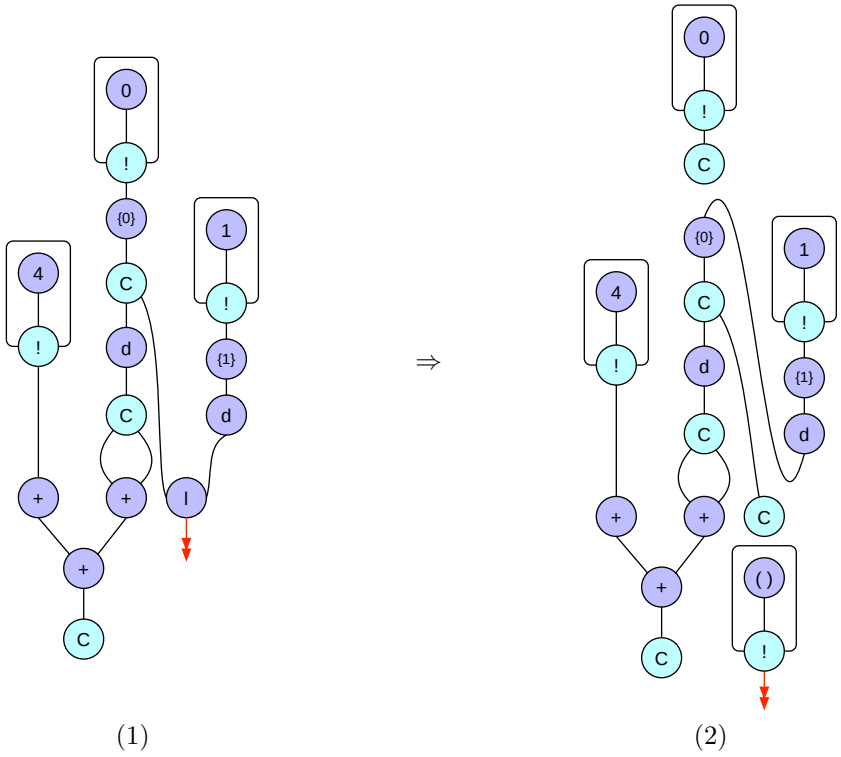
be reduced (e.g. $2+2$ to 4) while sharing all sub-terms that depend on cells, which might subsequently change their values.

This simple example illustrates the benefit of the DGoIM machine, which is not just an operational semantics but an abstract machine semantics with a precise cost model. The graphs (4-7) are all equivalent extensionally, as they will compute the same value. The difference between them is one of cost, as (4) has smaller size (memory footprint), (6) evaluates faster, and (7) is both smaller and faster.

Linking The value of the dataflow cells can be changed in two ways, by linking or by assignment. Linking is a more general operation which changes the dependencies between dataflow graphs, whereas assignment only changes the current value of a cell. Consider a small elaboration on the previous example:[4]

$$\texttt{let}\ c = \texttt{ref}\ 0\ \texttt{in}\ (\lambda x.\lambda y.(y+y)+(x+x))\ (\texttt{deref}\ c)\ 2;\ \texttt{link}\ c\ (\texttt{deref}\ (\texttt{ref}\ 1))$$

[4] `https://cwtsteven.github.io/TSD-visual/?ex=link`

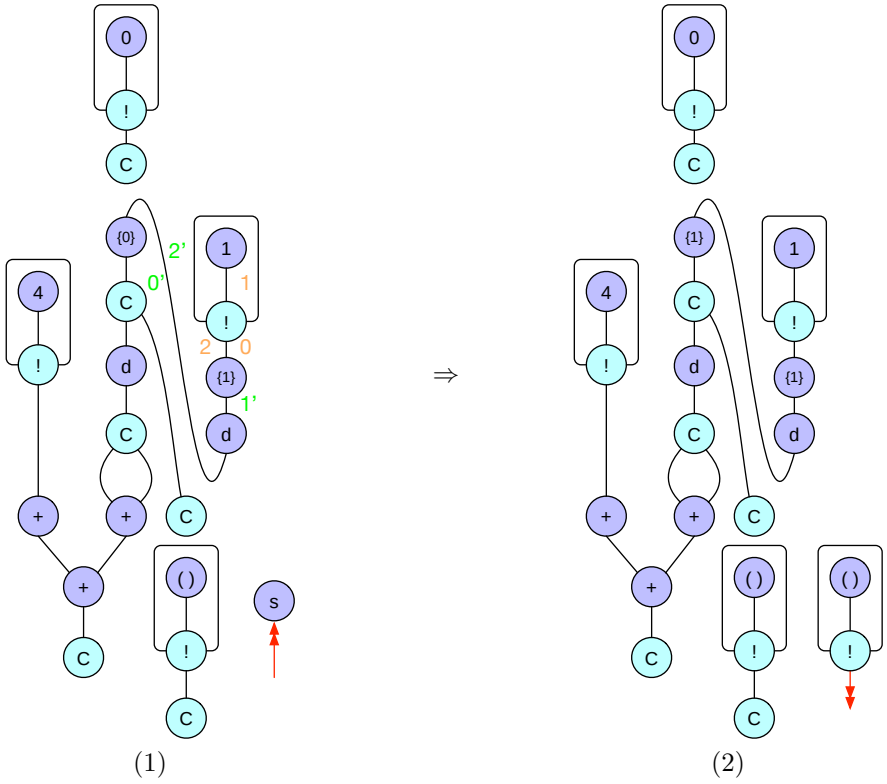


(1) (2)

Figure 3.5: Linking

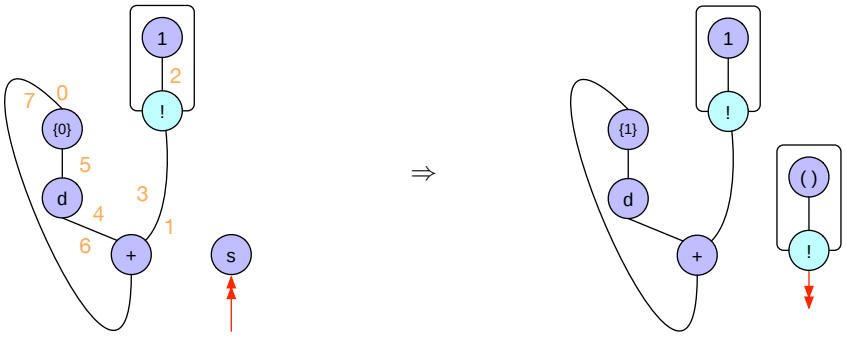
The execution of the 'link' command corresponds to the rewrite of the dataflow graph in Fig. 3.5. The link computation-graph node (l-node) will cause its argument (cell with value 1) to be set as the dependency of the cell with current value 0, previously dependent only on the constant 0.

Step propagation Propagation of changes occurs whenever the command `step` is issued (the s-node in Fig. 3.6). In this case, all cells produce a special propagation token which re-evaluates its dependencies (following the green and brown paths) and updates the current value. In the example, this results in changing the value of the cell holding 0 to it holding 1. Since the only rewrite during propagation is when all tokens are back to the cells to update their content, they can be run concurrently because a cell will only be updated once by one token. Also, since each token does not transit from a cell's in-going edge to its out-going edge; therefore dependency loops, which can only occur via cells, will always terminate. For



(1) (2)

Figure 3.6: Step propagation



(1) (2)

Figure 3.7: Step propagation with cycle

example, consider the following term

$$\texttt{let}\ c = \texttt{ref}\ 0\ \texttt{in}\ \texttt{link}\ c\ (\texttt{deref}\ c + 1);\ \texttt{step}$$

where the corresponding graph is shown in Fig. 3.7. The token that re-evaluates the dependency of c is never running into an infinite loop. This will be formally proved in section 6.8.

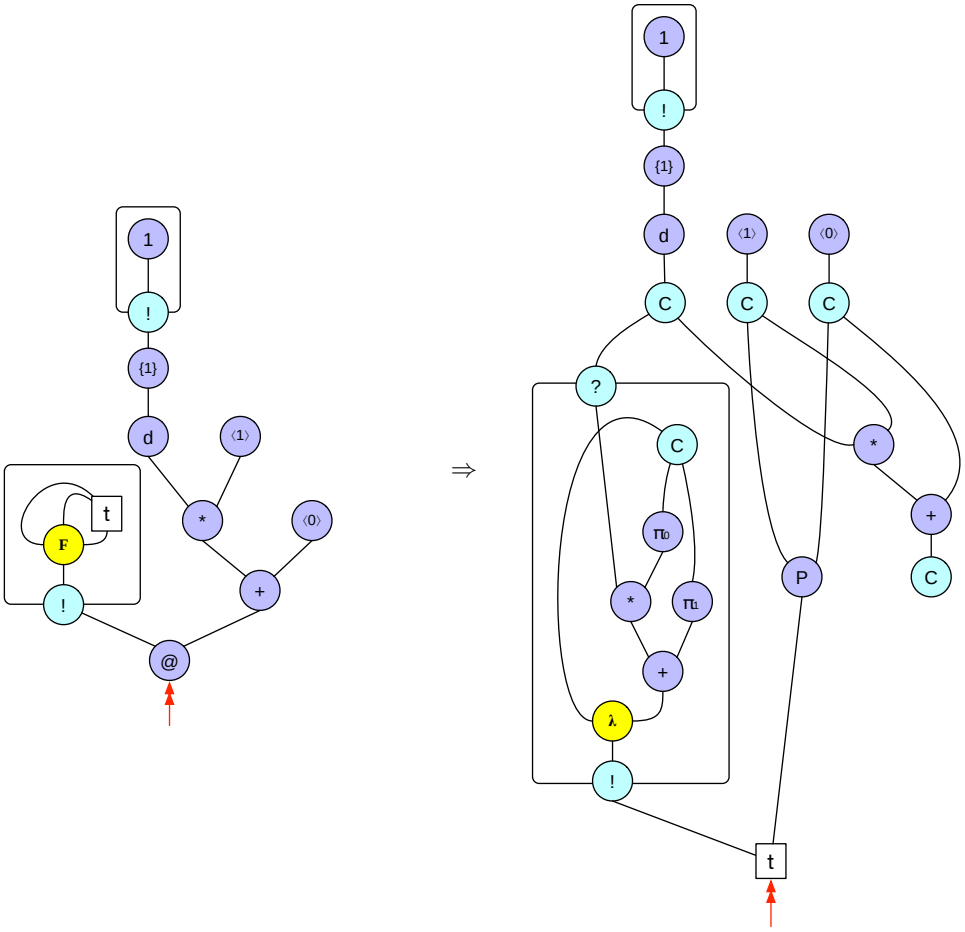


Figure 3.8: Fusion

Provisional constants and fusion Finally, provisional constants can be fused by using the fusion binder. Let us consider the following program:

$$\texttt{let}\ x = \texttt{ref}\ 1\ \texttt{in}\ \texttt{let}\ m = \texttt{deref}\ x * \texttt{pc}\ 1 + \texttt{pc}\ 0\ \texttt{in}\ (\mathcal{F}(f, x).t)\, m$$

Provisional constants are created by using the `pc` operation, which takes a number as an argument. The corresponding graphs right before and after fusion are shown in Fig. 3.8. The provisional constants `pc` 1 and `pc` 0 are denoted by the $\langle 1 \rangle$-node and the $\langle 0 \rangle$-node. The fusion binder is denoted by the F-node with two feedback edges representing the bound variables f and x. When applied to the model, it will create two sub-graphs: (1) a 'skeleton' function that takes a vector as an argument and returns a dataflow graph that has the same structure

as the original model where the π_i-node is the projection of the ith element of a vector; and (2) a parameter node P-node connecting to all the provisional constants in the model. Notice that the created function keeps the data dependencies by only sharing but not copying the input cell $\{1\}$ and thus when the function is applied more than once they all share the same input cell.

There is a subtlety on what should be copied or shared. For example, if the program is as follows:

$\texttt{let}\ x = \texttt{ref}\ 1\ \texttt{in}\ \texttt{let}\ y = \texttt{ref}\ (\texttt{deref}\ x{+}1)\ \texttt{in}\ \texttt{let}\ m = \texttt{ref}\ (\texttt{deref}\ y{+}1)\ \texttt{in}\ (\mathcal{F}(f,x).\ldots)\ m$

then the cells y and m will be copied into f as well in order to preserve the structure of the model. In other words, only cells that do not depend on other cells are shared. This also means that any subsequent commands acting on m will not affect other copies of m. For example, in the code

$$\begin{aligned}
&\texttt{let}\ x = \texttt{ref}\ 1\ \texttt{in}\\
&\texttt{let}\ y = \texttt{ref}\ (\texttt{deref}\ x + 1)\ \texttt{in}\\
&\texttt{let}\ m = \texttt{ref}\ (\texttt{deref}\ y + 1)\ \texttt{in}\\
&(\mathcal{F}(f,x).\texttt{let}\ n = f\ ps\ \texttt{in}\ \texttt{link}\ y\ \texttt{deref}\ x + \texttt{deref}\ x)\ m
\end{aligned}$$

the `link` operation will only modify the structure of m but not n because there is no variable pointing to the copies of y that can be used by the programmer.

Chapter 4

Programming in TSD

In this section, we will describe several typical example programs written in TSD. This includes some regression based examples such as linear regression (section 4.1) and neural networks (section 4.2) and examples that try to find the optimal system parameters, e.g. the initial population of an eco-sphere (section 4.3) or the shape of a motorway (section 4.4).

4.1 Linear regression[1]

Let us first revisit the motivating example in the introduction, *linear regression* with stochastic gradient descent. But before that, we need to first express the gradient descent algorithm in TSD.

4.1.1 Gradient descent[2]

Gradient descent is one of the most common ways to optimise a model. It is an iterative algorithm for finding the local minimum value of a function, e.g. for a function $y = f(x)$ we want to know the value of x when y is at minimum. The idea is that if we know the derivative of the function, $f'(x)$, then finding the local minimum of $f(x)$ is the same as finding the value of x when $f'(x) = 0$. We start from a random point at $f'(x_0)$, if $f'(x_0) > 0$, it means that

[1] https://github.com/cwtsteven/Transparent-Synchronous-Dataflow/tree/master/examples/linear_regression
[2] https://github.com/cwtsteven/Transparent-Synchronous-Dataflow/tree/master/examples/lib

its slope or gradient is positive and $f(x_0)$ is on the right-hand side of the local minimum; otherwise if $f'(x_0) < 0$ then $f(x_0)$ is on the left of the minimum. In these cases, we either decrease or increase x_0 to get x_1 respectively and calculate the gradient again until we get $f'(x_n) = 0$ or within a tolerable range.

However, in certain situations, the derivative function $f'(x)$ might be unknown. In this case, the gradient can be calculated using numerical differentiation as follows:

$$f'(x) = \lim_{\Delta_x \to 0} \frac{f(x + \Delta_x) - f(x)}{\Delta_x} \tag{4.1}$$

In practice, we set Δ_x to a very small value to calculate the gradient of a particular point x_0.

The gradient descent function `grad_desc` in TSD is defined in Fig. 4.1. `grad_desc` takes three arguments: `ps:Prm a` is the fused parameters of the model, `grad` is the gradients graph, and `rate:`$\mathbb{F}$ is the learning rate. It returns a function `optimise` that calls the function `forward` and then add the gradients to the parameters `ps` multiplied by the learning rate. The function `forward` represents the feed-forward procedure of the model.

```
(* grad_desc : ∀a. Prm a -> Vec a -> 𝔽 -> (𝔽 -> 𝔽) -> 𝔽 *)
let grad_desc = λps.λgrad.λrate.
  let optimise = λforward.
    forward 0; (* '0' is just a dummy argument, it has no special meaning *)
    assignp ps (peekp ps ⊞ (peek rate ⊠ peekp grad))
  in
  optimise
in
```

Figure 4.1: Gradient descent

Numerical differentiation `num_grad` is defined in Fig. 4.2. `num_grad` takes four arguments: `f:`$Vec\ a \to \mathbb{F}$ is the skeleton function of the model, `ps:`$Prm\ a$ is the fused parameters, `loss` is the loss function that is going to be minimised and `delta` corresponds to Δ_x in the equation. `num_grad` makes use of the `foldp` operation to calculate the gradients for each dimension in the parameters `ps`. Firstly, it creates a graph that calculates the original loss `oldloss` using `f` and `ps`. Secondly, it uses `foldp` to apply a function to each dimension of the parameters that creates a new loss `newloss` by adding `delta` to the dimension, and

calculates the gradient using equation 4.1. `foldp` will return a big graph that calculates all the gradients in every dimension of `ps`.

```
(* num_grad : ∀a. f : (Vec a -> 𝔽) -> ps : Prm a
              -> loss : ((Vec a -> 𝔽) -> Vec a -> 𝔽) -> delta : 𝔽 -> Vec a
*)
let num_grad = λf.λps.λloss.λdelta.
  let oldloss = loss f (derefp ps) in
  foldp (λe.
            let newloss = loss f ((delta ⊠ e) ⊞ derefp ps) in
            ((oldloss - newloss) / delta) ⊠ e) ps
in
```

Figure 4.2: Numerical gradients graph

4.1.2 Using gradient descent

```
(* mse : 𝔽 -> 𝔽 -> 𝔽 *) (* mean squared error *)
let mse = λy.λy_exp. (y - y_exp) * (y - y_exp) in

(* main program *)
let x = ref 0 in
let y = deref x * pc 1 + pc 0 in
let y_exp = ref 0 in
(* `fuse (f,ps) from y in t' is a syntatic sugar of `(F(f,ps). t) y' *)
fuse (f, ps) from y in
let mseloss = λf.λv.mse (f v) (deref y_exp) in
let grad = num_grad f ps mseloss 0.01
let optimise = grad_desc ps grad 0.1 in
(* '0' is just a dummy return value, it has no special meaning *)
optimise (λ_. assign x 1; assign y_exp 10; 0);
optimise (λ_. assign x 2; assign y_exp 14; 0);
optimise (λ_. assign x 3; assign y_exp 16; 0);
```

Figure 4.3: Linear regression

A linear regression example written in TSD is shown in Fig. 4.3. Note that `fuse (f,ps) from y in t` is simply a syntactic sugar of $(\mathcal{F}(f, ps).t)\ y$. In the main program, we construct a simple model with input x and output y, such that $y = ax + b$. The model is stateful and is initialised to 0 via a call to a `ref` constructor function. The parameters are initialised such that $a = 1, b = 0$, and are indicated as 'provisional constants' (`pc`). This means that

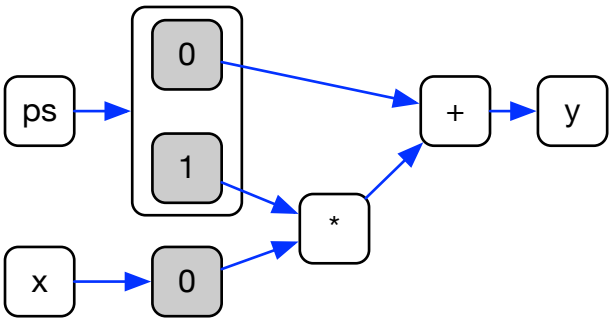


Figure 4.4: Graphical model of linear regression

in the course of evaluation they behave like constants, but they can be '*fused*' into a vector of parameters (`ps`) which can be optimised as a whole. *y_exp* is the cell that is going to store the expected output used for training. An informal schematic of the resulting model is in Fig. 4.4. The stateful components, cells and provisional constants, are marked as grey. After constructing the model we can then fuse the parameters which also creates the skeleton function of this model.

Line 11 creates the gradients graph using `num_grad`. The resulting gradient graph is shown in Fig. 4.5. The sub-graph with the purple arrows indicates the calculation to `oldloss`, the original loss, while the one with blue arrows represents that of the `new_loss`, the loss obtained after moving a small step in `a`. `newloss` is created for each dimension in the parameters, and the output represents the corresponding partial derivatives. Line 14 to 16 call the `optimise` function by pushing data into `x` and `y_exp`. Notice that each sample data is pushed manually because data structures are not present in the language yet, but we assume it is not difficult to extend the language.

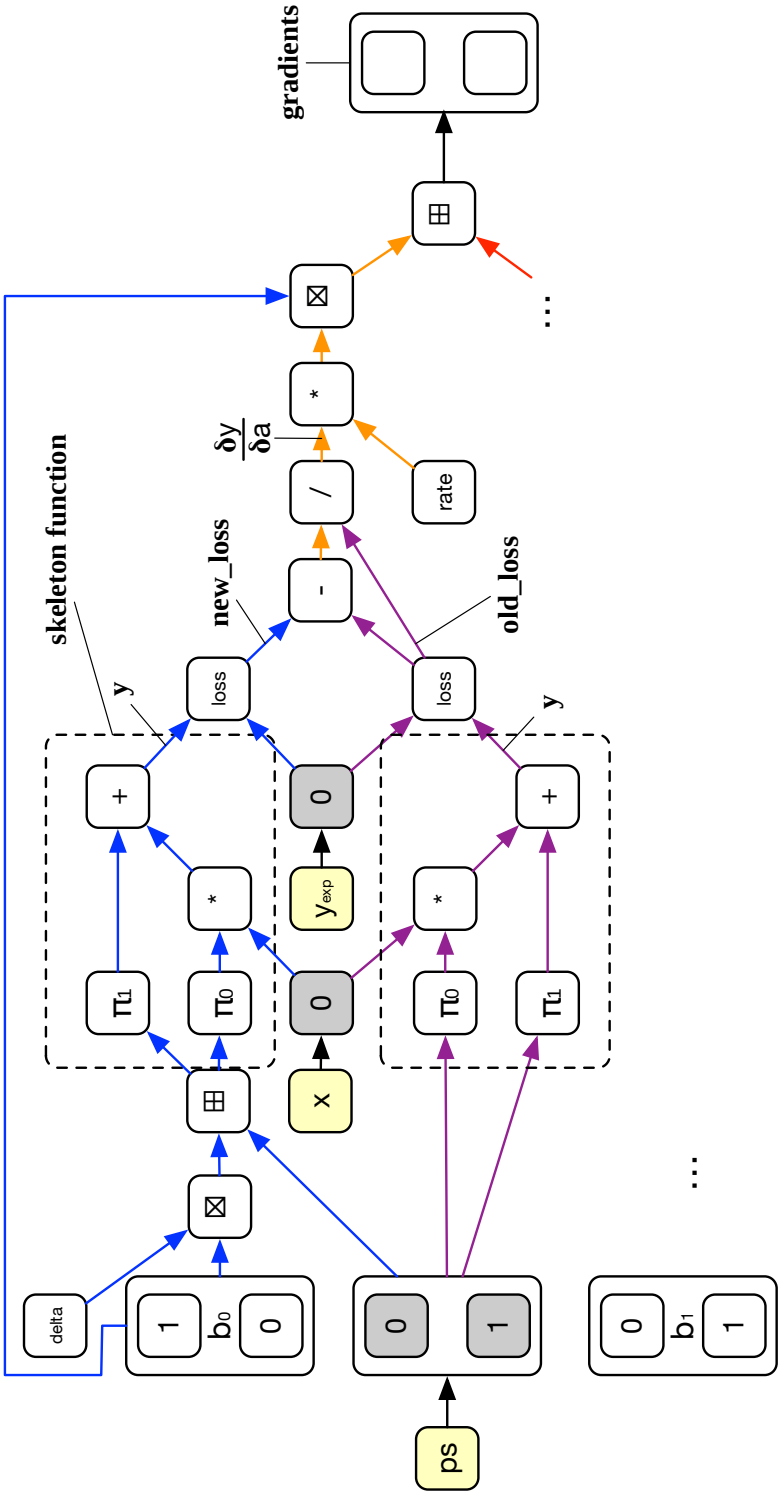


Figure 4.5: Numerical gradients graph for the linear regression example

4.1.3 Parallelisation and pipelines

The example above does not make use of the parallel propagation mechanism provided by the language, as the gradients of each dimension are calculated sequentially. We can introduce parallelism by putting cells into the graph. For example, we can use a slightly different mean squared error function as follows:

```
let mse' = λy.λy_exp. ref ((y - y_exp) * (y - y_exp)) in

let mseloss' = λf.λv.mse' (f v) (deref y_exp) in

let grad = num_grad f ps mseloss' 0.01 in
let optimise = grad_desc ps grad 0.1 in
optimise (λ_. assign x 1; assign y_exp 10; step; 0);
optimise (λ_. assign x 2; assign y_exp 14; step; 0);
optimise (λ_. assign x 3; assign y_exp 16; step; 0);
```

As shown in the above code snippet, we add a cell when the loss function is called. Therefore, there are now two layers of cells in the created gradient graph: 1) the input cells `x` and `y_exp`; and 2) the cells that store the loss of each dimension in `ps`. Therefore, we need to modify the `forward` function to add an extra `step` in order to propagate the changes in the inputs into the second layer. When the `step` command is called, the loss of each dimension will be calculated in parallel.

For graphs that have simple regular structure (like in this example), it is easy to see how many layers there are; and as such it is not difficult to predict how many `step` commands are needed to propagate changes through the graph. But sometimes programmers will need to design the graph layout carefully in order to synchronise the output of two graphs in order to create the intended dataflow. This will be discussed in more details in section 4.5 when we introduce another example.

By putting cells into the gradient graph, we can also define batch gradient descent with pipeline parallelism easily in TSD. Assuming that we use the average loss for each batch, then the program can be written as follows:

```
let accloss = λloss.λf.λv.
  let s = ref 0 in
  link s (s + loss f v); deref s
in

let grad = num_grad f ps (accloss mseloss) 0.01 in
let optimise = grad_desc ps grad 0.1 in
optimise (λ_. assign x 1; assign y_exp 10; step;
              assign x 2; assign y_exp 14; step;
              assign x 3; assign y_exp 16; step;
              0
         );
```

`accloss` creates a cell initialised with 0 accumulating the `loss` of all the data points which in turn creates a pipeline in the graph such that when `step` is executed, the current data point is pushed to the input cells while the loss for the previous data point is calculated simultaneously. In such a way, TSD provides fine-grained controls over both operation and pipeline parallelism.

After iterating a batch and changing the parameters, we can either (1) create a new gradient graph for the next batch or (2) reset all the created cells. The former could be expensive and redundant if the model is static. The latter can be achieved by adding dependency to every cell such that they all depend on a `reset` signal and later setting it to true every time before iterating a batch, e.g.

```
let reset = ref false in

let accloss = λloss.λreset.λf.λv.
  let s = ref 0 in
  link s (if reset then 0 else s + loss f v); deref s
in

let grad = num_grad f ps (accloss (deref reset) mseloss) 0.01 in
let optimise = grad_desc ps grad 0.1 in
optimise (λ_. assign reset true; step;
              (* reset every cells depending on "reset" *)
              assign x 1; assign y_exp 10; step;
              assign x 2; assign y_exp 14; step;
              assign x 3; assign y_exp 16; step;
              0
         );
```

Notice that in the `forward` function, after line 10 is executed, the `reset` cell will be changed `false` again. This is because `assign` does not change its dependency and thus `reset` is still depending on the constant `false` and after a `step` it will be changed back to it.

This way of adding a `reset` stream is not ideal because imagine having after writing a system with a lot of cells, adding a `reset` stream to the system would require every cell to be modified like the `accloss` in line 3. An alternative approach would be to provide a `reset` command which will reset all the cells to their initial values. It is a very common approach in languages that have features for creating automata, such as Lucid Synchrone and Zélus. This extension would not be difficult to implement as when every cell are created they can simply remember their initial values and when `reset` is called they are assigned the initial value. For simplicity, we will leave it to future development.

A full example of batch gradient descent is shown in Fig. 4.6.

```
let mse = λy.λy_exp. (y - y_exp) * (y - y_exp) in

let train = λoptimise.λforward.
  rec h. λn.
    if n <= 0 then 0 else optimise forward; h (n - 1)
in

let reset = ref false in

let accloss = λloss.λreset.λf.λv.
  let s = ref 0 in
  link s (if reset then 0 else s + loss f v); deref s
in

let x = ref 0 in
let y = deref x * pc 1 + pc 0 in
let y_exp = ref 0 in
fuse (f, ps) from y in
let mseloss = λf.λv.mse (f v) (deref y_exp) in
let grad = num_grad f ps (accloss (deref reset) mseloss) 0.01 in
let optimise = grad_desc ps grad 0.1 in
let forward = λ_.
  assign reset true; step;
  assign x 1; assign y_exp 10; step;
  assign x 2; assign y_exp 14; step;
  assign x 3; assign y_exp 16; step;
  0
in
train optimise forward 1000;
assign x 100; step; peek y
```

Figure 4.6: Full example of linear regression with batch gradient descent

4.2 Recurrent neural networks[3]

```
(* input_layer : 𝔽 -> List Cell *)
let input_layer = rec h. λn.
  if n <= 0 then [] else ref 0 :: h (n - 1)
in

(* perceptron : (List 𝔽 -> 𝔽) -> List 𝔽 -> 𝔽 *)
let perceptron = λf.λinp.
  let create_pcs = rec h. λn.
    if n <= 0 then [] else pc 0 :: h (n - 1)
  in
  let pcs = create_pcs (length (peek inp)) in
  let sum = dot_product inp pcs + pc 0 in (* adding the bias *)
  f sum
in

(* dense_layer : (List 𝔽 -> 𝔽) -> List 𝔽 -> 𝔽 -> List 𝔽 *)
let dense_layer = λf.λinp. rec h. λn.
  if n <= 0 then [] else f inp :: h (n - 1)
in

(* reflist : List 𝔽 -> List Cell *)
let reflist = rec h. λinp.
  if inp == [] then [] else ref (head inp) :: h (tail inp)
in

(* dereflist : List Cell -> List 𝔽 *)
let dereflist = rec h. λinp.
  if inp == [] then [] else deref (head inp) :: h (tail inp)
in
```

Figure 4.7: Helper functions for creating neural networks

In this example, we will show how recurrent neural networks can be implemented in TSD. For the purpose of convenience, we will assume that lists are present in the language. We first define some helper functions for creating a neural network in Fig. 4.7. The function `input_layer` creates a list with `n` cells initialised with `0`. `perceptron` takes an activation function `f` and a list of inputs as arguments to create the appropriate number of provisional constants and returns their dot product. As its name suggests, it creates a perceptron in a neural network. `dense_layer` creates a dense layer of neurons by using the argument `f` and

[3] https://github.com/cwtsteven/Transparent-Synchronous-Dataflow/tree/master/examples/rnn

`inp`. `reflist` creates a list of cells from a list of $\mathbb{F}$ and `dereflist` deferences a list of cells. For example, a simple neural network can be constructed as follows:

```
let inp = input_layer 30 in
let hidden = reflist (dense_layer (perceptron sigmoid) (dereflist inp) 10) in
let out = reflist (dense_layer (perceptron tanh) hidden 2) in
```

The input layer has 30 dimensions, the hidden layer consists of 10 sigmoid perceptrons and the output layer contains 2 tanh perceptrons. This model can then be optimised using numerical gradient descent, as shown in the example of linear regression.

Since states and feedbacks are allowed in TSD we can also create a special kind of neural network, recurrent neural network (RNN), that has internal states and is aware of its own output. RNN is useful when the data is a sequence related to each other. It has found numerous applications such as time series prediction, speech recognition, grammar learning, etc. [55]

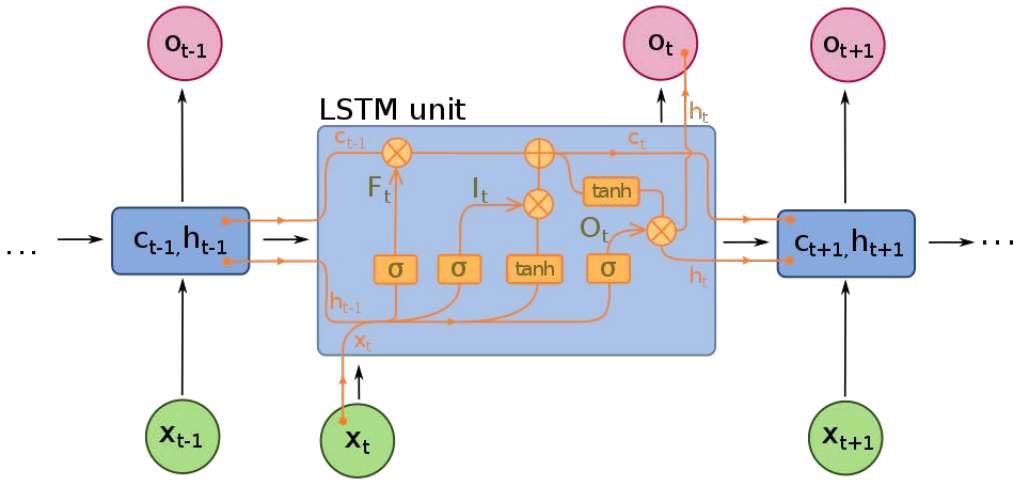


Figure 4.8: A LSTM unit[4]

In particular, we will show the implementation of a *long-short-term-memory* (LSTM) unit with a forget gate following this online tutorial[5]. Based on the dataflow shown in Fig. 4.8, a

[4]This diagram is extracted from `https://commons.wikimedia.org/wiki/File:Long_Short-Term_Memory.svg`.
[5]`https://colah.github.io/posts/2015-08-Understanding-LSTMs/`

LSTM cell can be represented as the following equations:

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f) \tag{4.2}$$

$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i) \tag{4.3}$$

$$\tilde{C}_t = tanh(W_{\tilde{C}} \cdot [h_{t-1}, x_t] + b_{\tilde{C}}) \tag{4.4}$$

$$C_t = f_t \cdot C_{t-1} + i_t \cdot \tilde{C}_t \tag{4.5}$$

$$o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o) \tag{4.6}$$

$$h_t = o_t \cdot tanh(C_t) \tag{4.7}$$

where x_t is the input signal, h_t is the output, f_t is the forget gate deciding whether the information of the state should be kept, C_t is the internal state, i_t and $\tilde{C}_t$ together are the new information to be added to the state and o_t is used to filter the output signal, Ws and bs are the weights and biases of each neuron. The corresponding code written in TSD is shown in Fig. 4.9.

```
(* lstm : 𝔽 -> List 𝔽 -> 𝔽 *)
let lstm = λreset.λinp.
  let sigmoid = λx. 1 / (1 + exp (-x)) in
  let h = ref 0 in let h_v = deref h_state in
  let c = ref 0 in let c_v = deref c_state in
  let f = perceptron sigmoid (h_v :: inp) in
  let i = perceptron sigmoid (h_v :: inp) in
  let c' = perceptron tanh (h_v :: inp) in
  link c_state (if reset then 0 else f * c_v + i * c');
  let o = perceptron sigmoid (h_v :: inp) in
  link h_state (if reset then 0 else tanh (root c_state) * o);
  h
in
```

Figure 4.9: LSTM in TSD

where `h`, `c`, `f`, `i`, `c'` and `o` correspond to h_t, c_t, f_t, i_t and o_t in the equations.

A simple LSTM network can be created as follow:

```
let reset = ref false in
let inp = input_layer 1 in
let hidden1 = dense_layer (lstm reset) (dereflist inp) 50 in
let hidden2 = dense_layer (lstm reset) hidden1 100 in
```

```
5  let out = reflist (dense_layer (perceptron (λx.x)) hidden2 1) in
```

4.3 Eco-sphere[6]

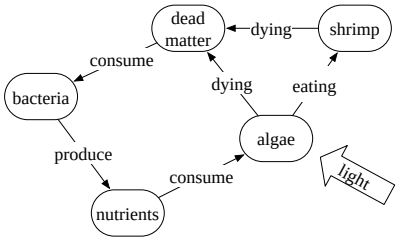


Figure 4.10: Eco-sphere model

This example shows the optimisation of a complex non-linear system with feedback loops. The problem is the construction of a self-sustaining *eco-sphere*, an airtight glass container containing water, nutrients, bacteria, algae, and shrimps. The only input to the system is light. Knowing the natural reproduction and death rates of bacteria, algae, and shrimps, as well as their nutritional needs a model of the system can be created, as seen in Fig. 4.10.

The model will be constructed as five interacting systems, one for each of the elements listed above. They will each have one state storing a positive value, which will go up or down depending on the interaction with the other relevant systems. The construction of the model is shown in Fig. 4.11. The `createModel` function initialises the provisional constants (tunable parameters) and the five state values with some arbitrary numbers; and establishes the connections between the systems using the `link` functions. Each of these cells are linked to their respective growth function, e.g. `alg_eqn` is the growth function of the algae which depends on the number of algae, nutrients and shrimps in the system. When the values of the dependencies change, the value of the linked cells are updated automatically, much like in a spreadsheet. Each cycle of updates will change each cell at most once, to prevent infinite propagations of changes.

[6] `https://github.com/cwtsteven/Transparent-Synchronous-Dataflow/blob/master/examples/ecoshpere.ml`

```
let createModel = λ_.
  let nut = ref 0 in  let alg = ref 0 in  let shr = ref 0 in
  let bac = ref 0 in  let ded = ref 0 in

  let init_n = pc 1000 in
  let init_a = pc 500 in
  let init_s = pc 10 in
  let init_b = pc 100 in
  let init_d = pc 1000 in

  link nut (nut_eqn init_n (deref nut) (deref alg) (deref bac) (deref ded));
  link alg (alg_eqn init_a (deref alg) (deref nut) (deref shr));
  link shr (shr_eqn init_s (deref shr) (deref alg));
  link bac (bac_eqn init_b (deref ded) (deref bac));
  link ded (ded_eqn init_d (deref bac) (deref alg) (deref nut) (deref shr));

  shr
in

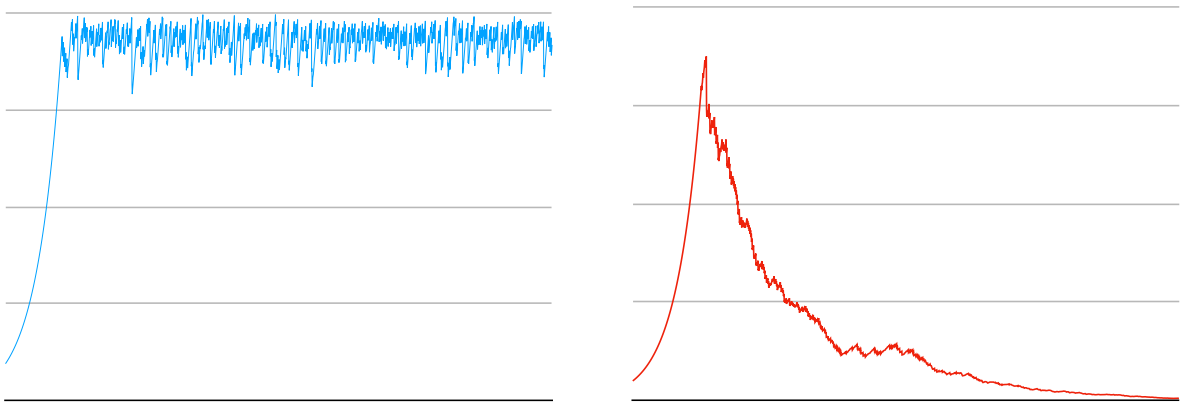
let stepn = rec f. λn.
  if n > 0 then step; f (n - 1) else ()
in

let shr = createModel () in
let visible = if shr > 0.5 then 1 else 0 in
let days_visible =
  let s = ref 0 in
  link s (if reset then 0 else s + visible); s
in

fuse (f, ps) from days_visible in
let loss = λf.λv. let days = f v in if days <= 0 then 1000000 else 1 / days in
let grad = num_grad f ps loss 0.1 in
let optimise = gradient_descent ps grad 0.1 in
train optimise (λ_. assign reset true; step; stepn 1000; 0) 1000;
```

Figure 4.11: Ecosphere: the model

With a `loss` function defining the design objective of maximising the overall number of shrimps over the lifetime of the eco-sphere (since they are the only visible creatures to its owner!) the optimisation of the initial proportions is similar to the programs above, emphasising the point that optimisation is system-independent. Fig. 4.12(a) shows a scenario that meets the design objective (the population of shrimp grows then fluctuates) and Fig. 4.12(b) shows one which does not (the population collapses eventually).



(a) Growth and fluctuation

(b) Growth and collapse

Figure 4.12: Shrimp population evolutions

4.4 Optimal road design[7]

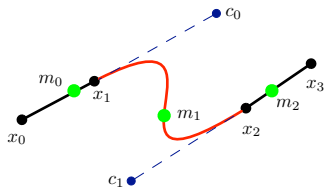


Figure 4.13: A 3-segment road model

In this more complex example, we shall consider optimising the design of a road. Each road is defined by a sequence of segments which alternate between straight lines and Bézier curves connecting the lines. The control points of the Bézier curves must lie along the straight segments, to ensure an overall smooth road. A car is assumed to *accelerate* along a straight line up to a point, then *decelerate*. On a curve, it will have the opposite behaviour, decelerating up to a point, then accelerating. At any point, if the angular velocity exceeds an allowed maximum, the car will crash. An example road with 3 segments, is given in Fig. 4.13. Given the endpoints of each segment x_i, given the maximum acceleration, deceleration, and angular velocity of the vehicle, the optimisation problem requires computing the Bézier control points c_i and the acceleration-break change points m_i, and the acceleration and deceleration along

[7] `https://github.com/cwtsteven/Transparent-Synchronous-Dataflow/blob/master/examples/road_optimisation.ml`

each segment such that the vehicle travels the length of the curve in minimum total time, without tipping over. The main program is given below.

```
let segs = createPrms segments in
let model = createModel segs in
fuse (f, ps) from model in
let grad = num_grad f ps (f ps) 0.01 in
let optimise = gradient_descent ps grad 0.1 in
train optimise (λ_. 0) 1000;
```

This model is stateless, it contains no `cells`. Given a list of straight segments (`segments`), the function `createPrms` will create *parameterised* segments where some data (the endpoints) are constants and other data (control points, acceleration and deceleration for each segment) are provisional constants (tunable variables). Then, the `createModel` function will create the mathematical model with all the details of computing the motion along the curves, given the accelerations and the control points. All the parameters of the model are collected into the vector `ps` using parameter `fusion`. The `loss` function for the purpose of optimisation is the overall time, which is read from the `model`, and, along with the parameter `ps` is passed to a `gradient_descent` optimiser. Following the final call the parameters are updated to their optimal values.

As an example, we tested with two straight lines having endpoints (in meters), $(0, 0), (1000, 0)$ and $(1500, 1000), (2500, 1000)$. The result is shown in Fig. 4.14.

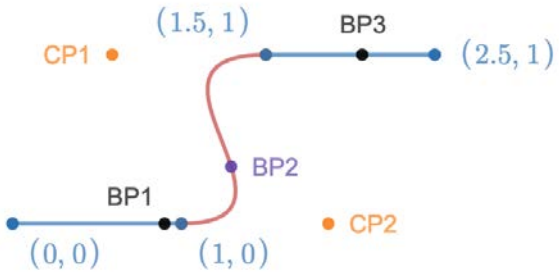


Figure 4.14: Optimised road (in kilometers)

The (blue) straight lines are the input segments. BPs are the break points in which the car should decelerate or accelerate in straight or Bézier segments respectively. CPs are the

control points of the (red) Bézier curve. The total amount of time used to cross the road is 69.84s.

What this example is meant to illustrate is the convenience of the `fusion` operation, which can extract any number of tunable parameters from a complex model, and aggregate them into a single vector which can then be optimised by a generic optimisation algorithm.

4.5 Sieve of Eratosthenes[8]

```
let fromn = λn. let s = ref n in link s (deref s + 1); deref s in
let filter = λinp.λn. inp == n or ((inp % n) <> 0) in
let inp = fromn 2 in
let sieve = ref (filter inp 2) in
let next = λ_. step; link sieve (root sieve and filter inp (peek inp)) in
let delay = ref inp in
let primes = if deref sieve then delay else 0 in
next 0; peek primes; // return 2
next 0; peek primes; // return 3
next 0; peek primes  // return 0
```

Figure 4.15: Sieve of Eratosthenes in TSD

In the final example, we compute a stream of prime numbers using the sieve of Eratosthenes. The code is in Fig. 4.15 and can be executed in the visualiser[9]. The function *fromn* takes an integer n and creates a stream of integers from n. The function *filter* takes an input stream *inp* and an integer n and creates a stream of booleans that could filter out the multiples of n (except for n itself) and anything smaller than n from *inp*. This filter function is meant to create the different layers of sieve that are used in generating prime numbers. Below are two examples:

[8] https://github.com/cwtsteven/TSD/blob/master/examples/primes.ml
[9] https://cwtsteven.github.io/TSD-visual?ex=primes

step	0	1	2	3	4	5	6	7	...
fromn 2	2	3	4	5	6	7	8	9	...
filter inp 2	t	t	f	t	f	t	f	t	...
filter inp 3	t	t	t	t	f	t	t	f	...

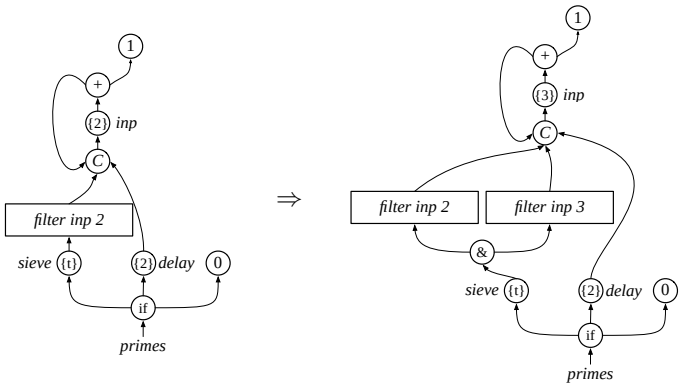
To implement the sieve of Eratosthenes, we first start with an input stream using *fromn* 2 (named *inp*). A cell is then initialised with the first layer *filter inp 2*. The sieve needs to expand by adding new layers of sieve in every step, without rewriting the dataflow graph during a step. The solution is to use the function *next* which unfolds a layer after each *step* as shown in Fig. 4.16. Finally, since *sieve* is a cell that depends on *inp* meaning that it is always one step behind *inp* and thus *primes = if sieve then inp else 0* would not give the correct stream. Instead, we need to create a delayed stream from *inp* (*delay = ref inp*) in order to synchronise the input of *primes*, i.e. *primes = if sieve then delay else 0*. Belows are the first few elements of each stream:

next	0	1	2	3	4	5	6	7	...
inp	2	3	4	5	6	7	8	9	...
sieve	t	t	t	f	t	f	t	f	...
delay	2	2	3	4	5	6	7	8	...
primes	2	2	3	0	5	0	7	0	...

This example also illustrates how graph dynamism is achieved in TSD. In between `steps`, the structure of the graph can be modified by using `link` and thus changing the data dependency during execution. In the above example, it was the structure of the sieve that was changed, as shown in Fig. 4.16. The ability to dynamically change the shape of graphs is useful as it gives the programmer the flexibility and freedom to create unconventional models. For instance, the sieve is impossible to create if the graph is static.

It is also interesting to contrast our code with the same program implemented in a similar language, Lucid Synchrone[10] shown in Fig. 4.17. The main distinction is that we use

[10] `https://www.di.ens.fr/~pouzet/lucid-synchrone/manual_html/manual015.html`



(a) before next() (b) after next()

Figure 4.16: Dataflow graph expansion for primes

imperative commands to set up the dataflow graph, whereas in Lucid Synchrone co-recursive stream equations are used for the same purpose. It is a matter of preference which one of the two versions are seen as more accessible.

```
let node first x = v
  where rec v = x fby v

let rec node sieve x =
  let clock filter = (x mod (first x))<> 0
  in  merge filter
            (sieve (x when filter))
            (true fby false)

let node from n = o where rec o = n fby (o + 1)

let clock sieve = sieve (from 2)

let node primes () = (from 2) when sieve
```

Figure 4.17: Sieve of Eratosthenes in Lucid Synchrone

Chapter 5

The TSD calculus

In this chapter, we describe formally the syntax and semantics of TSD.

5.1 Formal syntax

There are two sets of syntax, TSD and simTSD. The former is our source language with polymorphic names that would be transformed into the latter for execution. We will also show that the transformation is *sound* in Thm. 5.1.2, i.e. any well-typed term in TSD can be transformed into a well-typed term in simTSD.

5.1.1 TSD

Assume a set of *names* $a, b, ... \in \mathcal{Z}$, an ordered field $(\mathbb{F}, +, -, \times, \div, \leq)$, a $\mathcal{Z}$-indexed family of vector spaces *Vec* a and parameters *Prm* a. Let $\gamma ::= \mathbb{F} \mid \mathit{Cell} \mid \mathit{Unit} \mid \mathit{Vec}\ a \mid \mathit{Prm}\ a$. These are the base types. The language has simple types and type schemes with polymorphic names adapted from ML polymorphism as defined by the grammar $\tau ::= \gamma \mid \tau \to \tau$ (simple types), and $\sigma ::= \forall D.\tau$ (type schemes) where $D \subseteq \mathcal{Z}$. When $D = \emptyset$, we write just τ to

represent $\forall\{\}.\tau$ for simplicity. Terms are defined by the grammar below:

$$t ::= x \mid \lambda x.t \mid t\ t \mid n \mid op \mid \texttt{if}\ t\ \texttt{then}\ t\ \texttt{else}\ t \mid \texttt{rec}\ x.t \mid \texttt{pc}\ n \mid \mathcal{F}(f,x).t$$
$$\mid \texttt{letp}\ x = t\ \texttt{in}\ t$$

where *op* ranges over the following primitive operations:

$+, -, \times, \div, \leq : \mathbb{F} \to \mathbb{F} \to \mathbb{F}$	(field operations)
$\texttt{ref} : \mathbb{F} \to \mathit{Cell}$	(cell creation)
$\texttt{deref} : \mathit{Cell} \to \mathbb{F}$	(dereferencing of cells)
$\texttt{root} : \mathit{Cell} \to \mathbb{F}$	(dataflow graph of cells)
$\texttt{link} : \mathit{Cell} \to \mathbb{F} \to \mathit{Unit}$	(linking)
$\texttt{assign} : \mathit{Cell} \to \mathbb{F} \to \mathit{Unit}$	(assignment of cells)
$\texttt{peek} : \mathbb{F} \to \mathbb{F}$	(peeking)
$\texttt{step} : \mathbb{F}$	(step propagation)
$\boxplus : \mathit{Vec}\ a \to \mathit{Vec}\ a \to \mathit{Vec}\ a$	(vector addition)
$\boxtimes : \mathbb{F} \to \mathit{Vec}\ a \to \mathit{Vec}\ a$	(scalar multiplication)
$\boxdot : \mathit{Vec}\ a \to \mathit{Vec}\ a \to \mathbb{F}$	(dot product)
$\texttt{derefp} : \mathit{Prm}\ a \to \mathit{Vec}\ a$	(dereferencing of parameters)
$\texttt{assignp} : \mathit{Prm}\ a \to \mathit{Vec}\ a \to \mathit{Unit}$	(assignment of parameters)
$\texttt{peekp} : \mathit{Vec}\ a \to \mathit{Vec}\ a$	(peeking of vectors)
$\texttt{foldp} : (\mathit{Vec}\ a \to \mathit{Vec}\ a) \to \mathit{Prm}\ a \to \mathit{Vec}\ a$	(folding over basis)

Table 5.1: Primitive operations

with $\{\boxplus, \boxtimes, \boxdot, \texttt{derefp}, \texttt{assignp}, \texttt{peekp}, \texttt{foldp}\}$ being overloading operations with arbitrary names.

The core of the language is PCF [69] extended with *cells* (`ref` t), provisional constants (`pc` n), a fusion binder ($\mathcal{F}(f,x).t$) and a set of $\mathcal{Z}$-indexed operations *op* of various arities. The type $\mathbb{F}$ denotes either a normal value or a dataflow graph, where its current value can be returned as a constant (`peek`). Cells are created using `ref` and can be used to create a graph using `deref`. They can also be assigned to (`assign`) just like states. `root` returns the dataflow dependency of a cell. Active dependencies can be established between cells (`link`). Any changes to a dependency are automatically propagated when the `step` function is called. Provisional constants (`pc` n) can be fused by the fusion binder into a vector of parameters

$$\frac{A \vdash \Gamma, x : \forall D.\tau \quad B \subseteq A}{A \mid \Gamma, x : \forall D.\tau \vdash x : \tau[{}^{B}/_{D}]}\ \mathit{var} \qquad \frac{A \mid \Gamma, x : \forall\{\}.\tau \vdash t : \tau'}{A \mid \Gamma \vdash \lambda x.t : \tau \to \tau'}\ \mathit{abs}$$

$$\frac{A \mid \Gamma \vdash t' : \tau \to \tau' \quad A \mid \Gamma \vdash t : \tau}{A \mid \Gamma \vdash t'\ t : \tau'}\ \mathit{app} \qquad \frac{A \vdash \Gamma}{A \mid \Gamma \vdash n : \mathbb{F}}\ \mathit{const} \qquad \frac{A \vdash \Gamma}{A \mid \Gamma \vdash \mathit{op} : \tau}\ \mathit{op}$$

$$\frac{A \mid \Gamma \vdash t : \mathbb{F} \quad A \mid \Gamma \vdash t_1 : \gamma \quad A \mid \Gamma \vdash t_2 : \gamma}{A \mid \Gamma \vdash \texttt{if}\ t\ \texttt{then}\ t_1\ \texttt{else}\ t_2 : \gamma}\ \mathit{if}$$

$$\frac{A \vdash \Gamma, \tau \to \tau' \quad A \mid \Gamma, f : \tau \to \tau' \vdash t : \tau \to \tau'}{A \mid \Gamma \vdash \texttt{rec}\ f.t : \tau \to \tau'}\ \mathit{rec} \qquad \frac{A \vdash \Gamma}{A \mid \Gamma \vdash \texttt{pc}\ n : \mathbb{F}}\ \mathit{prov}$$

$$\frac{a, A \mid \Gamma, f : \mathit{Prm}\ a \to \mathbb{F}, x : \mathit{Prm}\ a \vdash t : \tau \quad a \# \tau \quad a \notin A}{A \mid \Gamma \vdash \mathcal{F}(f, x).\ t : \mathbb{F} \to \tau}\ \mathit{fusion}$$

$$\frac{A \mid \Gamma, f : \forall D.\tau \to \tau' \vdash u : \tau'' \quad A \cup D \mid \Gamma, x : \tau \vdash t : \tau' \quad D = \mathit{names}(\tau)/A}{A \mid \Gamma \vdash \texttt{letp}\ f = \lambda x.t\ \texttt{in}\ u : \tau''}\ \mathit{letp}$$

Figure 5.1: Typing rules of TSD

that can be assigned to (`assignp`). The operations are sometimes written in infix notation, as usual. Sometimes $+\ t\ u$ is also used for $t\ +\ u$. Generalising cells to storing arrays or arrays of arrays ('*tensors*') is possible. It is not conceptually challenging, but it introduces extra bureaucracy, which is besides the point here.

The typing rules are defined in Fig. 5.1. We say that a subset of $\mathcal{Z}$, A, supports Γ, denoted by $A \vdash \Gamma$, when all the *free* names appearing in Γ are also in A. A type judgement is in the form of $A \mid \Gamma \vdash t : \tau$ where A represents the scope of usable names; and $\Gamma = x : \sigma, y : \sigma', \ldots$ is the context represented as a mapping from variables to type schemes. The var-rule allows a type scheme to be instantiated to simple types. The abs-rule, app-rule, const-rule are usual. The op-rule represents all the rules listed in Table. 5.1. The if-rule however differs from the usual setting and only allows the branches to have ground types. This is because if functions are allowed to appear in the branches, then the dataflow graph may not be pure and thus leads to unsafe behaviour. For example, consider the function

$$\texttt{let}\ f = \texttt{if}\ z <= 3\ \texttt{then}\ \lambda x.\texttt{assign}\ x\ 10\ \texttt{else}\ \lambda x.\texttt{assign}\ x\ 5$$

where z is a graph, applying f is a cell will create an impure dataflow graph with `assign`

inside it. Let us extend the example a bit more:

$$\texttt{let } f = \texttt{if } z <= 3 \texttt{ then } \lambda x.\texttt{assign } x\ 10 \texttt{ else } \lambda x.\texttt{assign } x\ 5 \texttt{ in}$$

$$\texttt{let } g = \texttt{if } z <= 3 \texttt{ then } \lambda x.\texttt{assign } x\ 9 \texttt{ else } \lambda x.\texttt{assign } x\ 4 \texttt{ in}$$

$$\texttt{let } a = \texttt{ref } 1 \texttt{ in}$$

$$\texttt{let } b = \texttt{ref}(f\ a; 1) \texttt{ in}$$

$$\texttt{let } c = \texttt{ref}(g\ a; 1) \texttt{ in}$$

During a `step`, it will result in a race condition when re-evaluating b and c as they both try to write to the cell a. Notice that we use $\texttt{let } x = t \texttt{ in } u$ in TSD only as the syntactic sugar of $(\lambda x.u)\ t$.

The rec-rule describes the usual recursion construct for call-by-value languages, where only recursive functions are allowed. This ensures that the evaluation of $\texttt{rec } f.e$ does not diverge. It is not a restriction on TSD but on call-by-value languages in general. Other functional languages such as OCaml and Scheme have different but similar restrictions [34, 66].

In the fusion-rule, we define the operation for gathering provisional constants as a variable binder $\mathcal{F}(f, x).t$ to prevent the name a from being used outside scope, which also leads to unsafe behaviour. For example, if we allow the name to be used outside scope, then in the term

$$(\lambda g.g\ (\texttt{pc } 1) \boxplus g\ (\texttt{pc } 2 + \texttt{pc } 3))(\mathcal{F}(f, x).\texttt{derefp } x)$$

$\mathcal{F}(f, x).\texttt{derefp } x$ will have type $\mathbb{F} \to \mathit{Vec}\ a$, and both $g\ (\texttt{pc } 1)$ and $g\ (\texttt{pc } 2 + \texttt{pc } 3)$ will have the same type $\mathit{Vec}\ a$ but actually live in different families of vector space, i.e. having different sizes. This restriction means that the return type of the fusion binder cannot contain its name. It is guarded by the condition $a\#\tau$. This rule is very similar to the *graph abstraction* rule in Idealised TensorFlow [62] but the semantics and the context of the language is different. For example, in TSD we differentiate between parameters and normal vectors where the former is assignable and fusion binds to an assignable vector of parameters instead of a vector. In

ITF, there is no such concept and thus parameters tuning is done via function application on a new vector. A more detailed discussion will be provided in chapter 8 when we compare TSD to various languages. The name *fusion* is chosen here simply because the its semantics is to fuse all the connecting provisional constants into a vector; and not to be confused with the fusion calculus [67] which is a concurrency model extending the π-calculus.

The `letp`-rule is the only rule that allows the creation of polymorphic functions. As discussed in the informal introduction, polymorphism is provided by syntactically copying codes; therefore a polymorphic term must be in canonical form, i.e. a value, or otherwise effects will be duplicated. In this case, it must be a lambda abstraction, since having polymorphic ground type value would lead to unsafe behaviour. For example, in $\mathcal{F}(g,y).\mathcal{F}(f,x).\texttt{letp}\ y = x\ \texttt{in}\ \ldots$ if y is polymorphic then it could be instantiated to other names, but it is actually pointing towards a vector in different families. Furthermore, the `letp`-rule also restricts any existing names in the current name scope to be considered polymorphic. This is because if they are allowed, then the following term would be well-typed. Assuming $x : \mathit{Prm}\ a$ and $y : \mathit{Prm}\ b$:

$$\mathcal{F}(f,x).\texttt{letp}\ f = \lambda z.z \boxplus \texttt{derefp}\ x\ \texttt{in}\ \mathcal{F}(g,y).f\ (\texttt{derefp}\ y); \ldots$$

This should be ruled out because x and y are parameters of different families. In the TSD type system this is not allowed since a is already in the name scope and thus $f : \mathit{Vec}\ a \to \mathit{Vec}\ a$ is not polymorphic; therefore $f\ (\texttt{derefp}\ y)$ would not be well-typed. Notice that we could have `letp` to be defined as the overloading version of `let` but for the sake of clarity we choose to separate the two.

5.1.2 simTSD

The syntax of simTSD is the same as that of TSD but instead the language has only simple types as defined by the grammar $\tau ::= \gamma \mid \tau \to \tau$. The typing rules are defined in Fig. 5.2.

$$\frac{}{\Gamma, x:\tau \vdash x:\tau}\ \mathit{var} \qquad \frac{\Gamma, x:\tau \vdash t:\tau'}{\Gamma \vdash \lambda x.t : \tau \to \tau'}\ \mathit{abs}$$

$$\frac{\Gamma \vdash t' : \tau \to \tau' \quad \Gamma \vdash t : \tau}{\Gamma \vdash t'\ t : \tau'}\ \mathit{app} \qquad \frac{}{\Gamma \vdash n : \mathbb{F}}\ \mathit{const} \qquad \frac{}{\Gamma \vdash \mathit{op} : \tau}\ \mathit{op}$$

$$\frac{\Gamma \vdash t : \mathbb{F} \quad \Gamma \vdash t_1 : \gamma \quad \Gamma \vdash t_2 : \gamma}{\Gamma \vdash \texttt{if}\ t\ \texttt{then}\ t_1\ \texttt{else}\ t_2 : \gamma}\ \mathit{if}$$

$$\frac{\Gamma, f : \tau \to \tau' \vdash t : \tau \to \tau'}{\Gamma \vdash \texttt{rec}\ f.t : \tau \to \tau'}\ \mathit{rec} \qquad \frac{}{\Gamma \vdash \texttt{pc}\ n : \mathbb{F}}\ \mathit{prov}$$

$$\frac{\Gamma, f : \mathit{Prm}\ a \to \mathbb{F}, x : \mathit{Prm}\ a \vdash t : \tau \quad a \# \tau}{\Gamma \vdash \mathcal{F}(f,x).\ t : \mathbb{F} \to \tau}\ \mathit{fusion}$$

Figure 5.2: Typing rules of simTSD

5.1.3 From TSD to simTSD

The transformation from TSD to simTSD is defined in Fig. 5.3. The transformation $\rho(t)_{\mathcal{B},\mathcal{P}}$ takes a term t in TSD as the argument together with $\mathcal{B}$ denoting the set of bound variables and $\mathcal{P}$ the set of polymorphic variables encountered so far; and returns a pair consisting of a term in simTSD, and a mapping $\mathcal{M}$ that maps polymorphic variables to a set of *fresh* variables. The mapping $\mathcal{M}$ denotes the set of new variables in simTSD corresponding to the same polymorphic variable in TSD.

The functions Ω and Ω' are defined as follows to ensure that every instance of a polymorphic variable are transformed into different variables:

$$\Omega(\mathcal{M}_t, \mathcal{M}_u) = \begin{cases} \mathit{true}, & \mathit{if}\ \forall x \in \mathit{dom}(\mathcal{M}_t \cup \mathcal{M}_u).\mathcal{M}_t(x) \cap \mathcal{M}_u(x) = \emptyset \\ \mathit{false}, & \mathit{otherwise} \end{cases}$$

$$\Omega'(\mathcal{M}_1, \mathcal{M}_2, \mathcal{M}_3) = \Omega(\mathcal{M}_1, \mathcal{M}_2) \wedge \Omega(\mathcal{M}_2, \mathcal{M}_3) \wedge \Omega(\mathcal{M}_3, \mathcal{M}_1)$$

The function $\Phi(\mathcal{M}_u(f), \texttt{let}\ f = \lambda x.t'\ \texttt{in})$ in the `let`-rule creates copies of $\texttt{let}\ f' = \lambda x.t'\ \texttt{in}$ for every variable $f' \in \mathcal{M}_u(f)$. For example,

$$\Phi(\{f', f''\}, \texttt{let}\ f = \lambda x.t'\ \texttt{in}) = (\texttt{let}\ f' = \lambda x.t'\ \texttt{in let}\ f'' = \lambda x.t'\ \texttt{in})$$

$$
\begin{array}{rcl}
\rho(x)_{\mathcal{B},\mathcal{P}} & = & \begin{cases} x,\ \emptyset & \mathit{if}\ x \notin \mathcal{P} \\ x',\ \{x \mapsto \{x'\}\} & \mathit{if}\ x \in \mathcal{P} \end{cases} \\
 & & \quad \mathit{where} \quad x' \quad \notin \mathcal{B} \\
\rho(\lambda x.t)_{\mathcal{B},\mathcal{P}} & = & \lambda x.t',\ \mathcal{M} \\
 & & \quad \mathit{where} \quad (t', \mathcal{M}) = \rho(t)_{\mathcal{B}\cup\{x\},\mathcal{P}} \\
\rho(t_1\ t_2)_{\mathcal{B},\mathcal{P}} & = & t'_1\ t'_2,\ \mathcal{M}_1 \cup \mathcal{M}_2 \\
 & & \quad \mathit{where} \quad (t'_1, \mathcal{M}_1) = \rho(t_1)_{\mathcal{B},\mathcal{P}} \\
 & & \qquad\quad (t'_2, \mathcal{M}_2) = \rho(t_2)_{\mathcal{B},\mathcal{P}} \\
 & & \qquad\quad \Omega(\mathcal{M}_1, \mathcal{M}_2) = \mathit{true} \\
\rho(n)_{\mathcal{B},\mathcal{P}} & = & n,\ \emptyset \\
\rho(op)_{\mathcal{B},\mathcal{P}} & = & op,\ \emptyset \\
\rho(\texttt{if}\ t\ \texttt{then}\ t_1\ \texttt{else}\ t_2)_{\mathcal{B},\mathcal{P}} & = & \texttt{if}\ t'\ \texttt{then}\ t'_1\ \texttt{else}\ t'_2,\ \mathcal{M} \cup \mathcal{M}_1 \cup \mathcal{M}_2 \\
 & & \quad \mathit{where} \quad (t', \mathcal{M}) = \rho(t)_{\mathcal{B},\mathcal{P}} \\
 & & \qquad\quad (t'_1, \mathcal{M}_1) = \rho(t_1)_{\mathcal{B},\mathcal{P}} \\
 & & \qquad\quad (t'_2, \mathcal{M}_2) = \rho(t_2)_{\mathcal{B},\mathcal{P}} \\
 & & \qquad\quad \Omega'(\mathcal{M}_1, \mathcal{M}_2, \mathcal{M}_3) = \mathit{true} \\
\rho(\texttt{rec}\ f.t)_{\mathcal{B},\mathcal{P}} & = & \texttt{rec}\ f.t',\ \mathcal{M} \\
 & & \quad \mathit{where} \quad (t', \mathcal{M}) = \rho(t)_{\mathcal{B}\cup\{f\},\mathcal{P}} \\
\rho(\texttt{pc}\ n)_{\mathcal{B},\mathcal{P}} & = & \texttt{pc}\ n,\ \emptyset \\
\rho(\mathcal{F}(f,x).\ t)_{\mathcal{B},\mathcal{P}} & = & \mathcal{F}(f,x).\ t',\ \mathcal{M} \\
 & & \quad \mathit{where} \quad (t', \mathcal{M}) = \rho(t)_{\mathcal{B}\cup\{f,x\},\mathcal{P}} \\
\rho(\texttt{letp}\ f = \lambda x.t\ \texttt{in}\ u)_{\mathcal{B},\mathcal{P}} & = & \Phi(\mathcal{M}_u(f), \texttt{let}\ f = \lambda x.t'\ \texttt{in})\ u', \mathcal{M}_t \cup (\mathcal{M}_u \setminus f) \\
 & & \quad \mathit{where} \quad (t', \mathcal{M}_t) = \rho(t)_{\mathcal{B}\cup\{x\},\mathcal{P}} \\
 & & \qquad\quad (u', \mathcal{M}_u) = \rho(u)_{\mathcal{B}\cup\{f\},\mathcal{P}\cup\{f\}} \\
 & & \qquad\quad \Omega(\mathcal{M}_t, \mathcal{M}_u) = \mathit{true}
\end{array}
$$

Figure 5.3: Transformation from TSD to simTSD

Consider the following term in TSD

$$\texttt{letp}\ h = \lambda x.x \boxplus x\ \texttt{in}\ \mathcal{F}(f,x).\mathcal{F}(g,y).\texttt{peekp}\ (h\ (\texttt{derefp}\ x)); \texttt{peekp}\ (h\ (\texttt{derefp}\ y))$$

Lets call this term T then we have that

$$\rho(T)_{\emptyset,\emptyset} = \Phi(\mathcal{M}_u(h), \texttt{let}\ h = \lambda x.t'\ \texttt{in})\ u', \mathcal{M}_t \cup (\mathcal{M}_u \setminus h)$$

where $(t', \mathcal{M}_t) = \rho(x \boxplus x)_{\{x\},\emptyset}$, $(u', \mathcal{M}_u) = \rho(u)_{\{h\},\{h\}}$ such that $\Omega(\mathcal{M}_t, \mathcal{M}_u) = \mathit{true}$. Continue with the procedure then $t' = x \boxplus x$, $\mathcal{M}_t = \emptyset$ and

$$u' = \mathcal{F}(f,x).\mathcal{F}(g,y).\texttt{peekp}\ (h_1\ (\texttt{derefp}\ x)); \texttt{peekp}\ (h_2\ (\texttt{derefp}\ y))$$

with $\mathcal{M}_u = \{h \mapsto \{h_1, h_2\}\}$. h_1 and h_2 come from the transformation of h since $h \in \mathcal{P}$. With all these sorted, we get

$$\begin{aligned}\rho(T)_{\emptyset,\emptyset} \equiv\ & \texttt{let}\ h_1 = \lambda x.x \boxplus x\ \texttt{in} \\ & \texttt{let}\ h_2 = \lambda x.x \boxplus x\ \texttt{in} \\ & \mathcal{F}(f,x).\mathcal{F}(g,y).\texttt{peekp}\ (h_1\ (\texttt{derefp}\ x)); \texttt{peekp}\ (h_2\ (\texttt{derefp}\ y))\end{aligned}$$

with $\mathcal{M}_T = \emptyset$.

As shown in this example, the transformation of a polymorphic function is sensitive to the number of times the function is used in the subsequent code. It is also very easy to show that this term is well-typed in simTSD since $h_1 : \mathit{Vec}\ a \to \mathit{Vec}\ a$ and $h_2 : \mathit{Vec}\ b \to \mathit{Vec}\ b$ assuming that $x : \mathit{Prm}\ a$ and $y : \mathit{Prm}\ b$.

5.1.4 Soundness of transformation

The soundness theorem is proved in Thm. 5.1.2 which states that any closed well-typed term in TSD can also be transformed into a closed well-typed term in simTSD. But first we need to define a relation between a typing context in TSD and the set of variables $\mathcal{B}$ and $\mathcal{P}$ used during the transformation (Def. 5.1.1); and a relation between the typing context of TSD and that of simTSD (Def. 5.1.2).

Definition 5.1.1. For any context Γ in TSD, any set of variables $\mathcal{B}$ and $\mathcal{P}$, we say that

$\Gamma, \mathcal{B}, \mathcal{P}$ is valid if they satisfies all the following conditions:

$$x \in \mathcal{B} \iff x \in dom(\Gamma)$$

$$x \in \mathcal{P} \implies \Gamma(x) = \forall D.\tau \wedge D \neq \emptyset$$

$$x \notin \mathcal{P} \implies \Gamma(x) = \forall \{\}.\tau$$

Definition 5.1.2. For any context Γ in TSD, any context Γ' in simTSD and any mappings $\mathcal{M}$, we say that $\Gamma \prec_{\mathcal{M}} \Gamma'$ if for any $x \in dom(\Gamma)$ where $\Gamma(x) = \forall D.\tau$, if $D = \emptyset$ then $x \in dom(\Gamma')$ and $\Gamma'(x) = \tau$ otherwise if $D \neq \emptyset$ then for any $x' \in \mathcal{M}(x)$, $x' \in dom(\Gamma')$ and $\Gamma'(x) = \tau[^{C}/_{D}]$ for some C.

Now we can prove that the transformation of any polymorphic functions instantiated with arbitrary names are also well-typed in simTSD.

Proposition 5.1.1. *For any well-typed term in TSD, $A \mid \Gamma \overset{TSD}{\vdash} t : \tau$ and any set $\mathcal{B}, \mathcal{P}$ such that $\Gamma, \mathcal{B}, \mathcal{P}$ is valid, let $(t', \mathcal{M}) = \rho(t)_{\mathcal{B},\mathcal{P}}$ then t' is also well-typed in simTSD i.e. there exists some Γ' such that $\Gamma' \overset{simTSD}{\vdash} t' : \tau$ and $\Gamma \prec_{\mathcal{M}} \Gamma'$.*

Proof. This is proved by induction on the type judgement $A \mid \Gamma \overset{TSD}{\vdash} t : \tau$.

Base case $A \mid \Gamma, x : \forall D.\tau \overset{TSD}{\vdash} x : \tau[B/D]$ If $x \notin \mathcal{P}$ then $D = \emptyset$ and thus $\rho(x) = (x, \emptyset)$. Let $\Gamma' = \{y : \tau' \mid y \in dom(\Gamma) \wedge y : \forall \{\}.\tau'\}$ then we have $\Gamma', x : \tau \overset{simTSD}{\vdash} x : \tau$ is well-typed such that $\Gamma, x : \forall \{\}.\tau \prec_{\emptyset} \Gamma', x : \tau$.

If $x \in \mathcal{P}$ then $D \neq \emptyset$ and $\rho(x) = (x', \{x \mapsto \{x'\}\})$. Let $\Gamma' = \{y : \tau' \mid y \in dom(\Gamma) \wedge y : \forall \{\}.\tau'\}$ then we have $\Gamma', x' : \tau[^{B}/_{D}] \overset{simTSD}{\vdash} x' : \tau[^{B}/_{D}]$ is well-typed such that $\Gamma, x : \forall D.\tau \prec_{\{x \mapsto \{x'\}\}} \Gamma', x' : \tau[^{B}/_{D}]$.

Base cases $A \mid \Gamma \overset{TSD}{\vdash} n : \mathbb{F}$**,** $A \mid \Gamma \overset{TSD}{\vdash} op : \tau$ **and** $A \mid \Gamma \overset{TSD}{\vdash} \mathtt{pc}\ n : \mathbb{F}$ These cases are trivial.

Inductive case $A \mid \Gamma \overset{TSD}{\vdash} \lambda x.t : \tau \to \tau'$ Since it is well-typed therefore we have that $A \mid \Gamma, x : \tau \overset{TSD}{\vdash} t : \tau'$. Because $\Gamma, \mathcal{B}, \mathcal{P}$ is valid therefore $(\Gamma, x : \forall \{\}.\tau), \mathcal{B} \cup \{x\}, \mathcal{P}$ is also

valid. As such, by using the induction hypothesis we have $\Gamma' \overset{simTSD}{\vdash} t' : \tau'$ for some Γ' where $(t', \mathcal{M}_t) = \rho(t)_{\mathcal{B}\cup\{x\},\mathcal{P}}$ such that $(\Gamma, x : \forall\{\}.\tau), \prec_{\mathcal{M}_t} \Gamma'$. As a result, $x \in dom(\Gamma')$ and $\Gamma(x) = \tau$. Finally, we have that $\Gamma' \setminus \{x\} \overset{simTSD}{\vdash} \lambda x.t' : \tau'$ and $\Gamma \prec_{\mathcal{M}_t} \Gamma' \setminus \{x\}$.

Inductive case $A \mid \Gamma \overset{TSD}{\vdash} \texttt{rec}\ f.t : \tau \to \tau'$ This case is the same as the above case (abstraction).

Inductive case $A \mid \Gamma \overset{TSD}{\vdash} \mathcal{F}(f, x).\ t : \mathbb{F} \to \tau$ This case is also similar to the abstraction case.

Inductive case $A \mid \Gamma \overset{TSD}{\vdash} t\ u : \tau'$ Since it is well-typed therefore we have that

$$A \mid \Gamma \overset{TSD}{\vdash} t : \tau \to \tau' \tag{5.1}$$

$$A \mid \Gamma \overset{TSD}{\vdash} u : \tau \tag{5.2}$$

Because $\Gamma, \mathcal{B}, \mathcal{P}$ is valid, by using the induction hypothesis on 5.1 and 5.2 we have that

$$\Gamma' \overset{simTSD}{\vdash} t' : \tau \to \tau' \tag{5.3}$$

$$\Gamma'' \overset{simTSD}{\vdash} u' : \tau \tag{5.4}$$

for some Γ', Γ'' where $(t', \mathcal{M}_t) = \rho(t)_{\mathcal{B},\mathcal{P}}$ and $(u', \mathcal{M}_u) = \rho(u)_{\mathcal{B},\mathcal{P}}$ such that

$$\Omega(\mathcal{M}_t, \mathcal{M}_u) = \mathit{true},\ \Gamma \prec_{\mathcal{M}_t} \Gamma'\ \mathit{and}\ \Gamma \prec_{\mathcal{M}_u} \Gamma''.$$

As a result, we know that for any $x \in \Gamma$ and $x : \forall D.\tau$, if $D = \emptyset$ then $x \in \Gamma'$ and $x \in \Gamma''$ such that $\Gamma'(x) = \Gamma''(x) = \tau$; otherwise if $D \neq \emptyset$ then because $\Omega(\mathcal{M}_t, \mathcal{M}_u) = \mathit{true}$, we have $\mathcal{M}_t(x) \cap \mathcal{M}_u(x) = \emptyset$. Finally, we can create a context $\Gamma^* = \Gamma' \cup \Gamma''$ such that $\Gamma \prec_{\mathcal{M}_t \cup \mathcal{M}_u} \Gamma^*$ and that $\Gamma^* \overset{simTSD}{\vdash} t'\ u' : \tau'$ is well-typed.

Inductive case $A \mid \Gamma \overset{TSD}{\vdash} \texttt{if}\ t\ \texttt{then}\ t_1\ \texttt{else}\ t_2 : \gamma$ This case is similar to the case above (application), as it can be seen as applying `if` to three arguments.

Inductive case $A \mid \Gamma \overset{TSD}{\vdash} \texttt{letp}\ f = \lambda x.t\ \texttt{in}\ u : \tau''$ Since it is well-typed therefore we have

$$A \cup D \mid \Gamma, x : \forall\{\}.\tau \overset{TSD}{\vdash} t : \tau' \tag{5.5}$$

$$A \mid \Gamma, f : \forall D.\tau \to \tau' \overset{TSD}{\vdash} u : \tau'' \tag{5.6}$$

where $D = names(\tau)/A$. Because we have that

$$\rho(\texttt{letp}\ f = \lambda x.t\ \texttt{in}\ u)_{\mathcal{B},\mathcal{P}} = (\Phi(\mathcal{M}_u(f), \texttt{let}\ f = \lambda x.t\ \texttt{in})\ \texttt{in}\ u', \mathcal{M}_t \cup (\mathcal{M}_u \setminus f))$$

where $(u', \mathcal{M}_u) = \rho(u)_{\mathcal{B}\cup\{f\},\mathcal{P}\cup\{f\}}$ and $(t', \mathcal{M}_t) = \rho(t)_{\mathcal{B}\cup\{x\},\mathcal{P}}$ such that $\Omega(\mathcal{M}_t, \mathcal{M}_u) = true$; therefore both $(\Gamma, f : \forall D.\tau \to \tau'), \mathcal{B} \cup \{f\}, \mathcal{P} \cup \{f\}$ and $(\Gamma, x : \forall\{\}.\tau), \mathcal{B} \cup \{x\}, \mathcal{P}$ are valid thus we can apply the induction hypothesis on 5.5 and 5.6 to get:

$$\Gamma' \overset{simTSD}{\vdash} t' : \tau' \tag{5.7}$$

$$\Gamma'' \overset{simTSD}{\vdash} u' : \tau'' \tag{5.8}$$

for some Γ', Γ'' such that $\Gamma, x : \forall\{\}.\tau \prec_{\mathcal{M}_t} \Gamma'$ and $\Gamma, f : \forall D.\tau \to \tau' \prec_{\mathcal{M}_u} \Gamma''$.

At this point, it suffices to show that for any $L \subseteq \mathcal{M}_u(f)$, $\Phi(L, \texttt{let}\ f = \lambda x.t'\ \texttt{in})\ u'$ is well-typed, i.e. $\Gamma^* \overset{simTSD}{\vdash} \Phi(L, \texttt{let}\ f = \lambda x.t'\ \texttt{in})\ u' : \tau''$ where $\Gamma^* = (\Gamma'' \setminus L) \cup (\Gamma' \setminus \{x\})$. This can be proved by induction on the number of elements $n < size(L)$ in L.

Base case when $n = 0$: then $L = \emptyset$ therefore we only need to prove that u' is well-typed which follows from 5.8.

Inductive case when $n = k{+}1$: let $L = \{f_{k+1}\} \cup L'$ and $N = (\lambda f_k \ldots (\lambda f_1.u')\ (\lambda x.t'))(\lambda x.t')$. Since $\Phi(L, \texttt{let}\ f = \lambda x.t'\ \texttt{in})\ u'$ is simply the syntactic sugar of

$$(\lambda f_{k+1}.N)\ (\lambda x.t')$$

by the induction hypothesis on k, there exists some Γ^* such that $\Gamma^* \overset{simTSD}{\vdash} N : \tau''$ and $\Gamma^* = (\Gamma'' \setminus L') \cup (\Gamma' \setminus \{x\})$. As such, we also have that $\Gamma^* \setminus \{f_{k+1}\} \overset{simTSD}{\vdash} \lambda f_{k+1}.N : (\tau \to \tau')[^C/_D] \to \tau''$ for some C.

From 5.7, we have that $\Gamma' \setminus \{x\} \overset{simTSD}{\vdash} \lambda x.t' : \tau \to \tau'$. Finally, by applying the app-rule of simTSD, we have that $\Gamma^* \setminus \{f_{k+1}\} \overset{simTSD}{\vdash} (\lambda f_{k+1}.N)\ (\lambda x.t') : \tau''$ and $\Gamma \prec_{\mathcal{M}_t \cup (\mathcal{M}_u \setminus f)} \Gamma^*$. □

Theorem 5.1.2 (Soundness of transformation)**.** *For any closed well-typed term in TSD,*

$- \mid - \overset{TSD}{\vdash} t : \tau$, *let* $(t', \mathcal{M}) = \rho(t)_{\emptyset,\emptyset}$ *then* t' *is also well-typed in simTSD, i.e.* $- \overset{simTSD}{\vdash} t' : \tau$.

Proof. This is simply a consequence of Prop. 5.1.1 taking $\Gamma = \mathcal{B} = \mathcal{P} = \emptyset$. □

5.2 Formal semantics

5.2.1 Graphs and translation

A term in simTSD is first translated into a graph and then executed by the graph abstract machine. In this section, we first define formally the graph and the translation.

A graph G is a set of nodes which are partitioned into *proper* nodes $Node_G$ (or just 'nodes') and *port* nodes $Port_G$ (or just 'ports'), and a set of directed edges connecting ports and nodes. Every node is associated and connected with several ports and each edge has at least one connecting port and there are no open edges. Every port is the source of at most one edge and the target of another. We also call the source and the target of a node the incoming port (or in-port) and the outgoing port (or out-port) respectively. For example, the +-node is connected to two out-ports and one in-port. We often denote a graph by $G : \Lambda \to \Gamma$ where Λ and Γ are the set of open incoming ports and outgoing ports respectively. They are also called the incoming and outgoing interfaces.

Ports are typed ($\theta ::= \nu \mid \kappa$), with either linear types $\nu ::= \mathbb{F} \mid \mathit{Unit} \mid \mathit{Vec}\ a \mid \kappa \multimap \kappa$ or qualified types $\kappa ::= !\nu \mid \mathit{Cell} \mid \mathit{Prm}\ a$. We also denote qualified base types as $\eta ::= !\mathbb{F} \mid !\mathit{Unit} \mid \mathit{Cell} \mid !\mathit{Vec}\ a \mid \mathit{Prm}\ a$.

We often refer to a node using their labels, for example, we say *a* λ*-node* for a node labelled by λ. Some labels are used to indicate nodes that interpret calculus constructs: λ-node (abstraction), @-node (application), n-node (constants), s-node (step), p, p_p-node (peek), d, d_p-node (dereference), r-node (root dependency), m-node (memory cell creation), \$-node (field operations), `if`-node (conditionals), μ-node (recursion), a, a_p-node (assignment), l-node (link), $\boxplus$-node (vector addition), $\boxtimes$-node (scalar multiplication), $\odot$ (dot product), f_p-node (folding), $\langle n \rangle$-node (provisional constants), F_a-node (fusion binder). Some nodes do

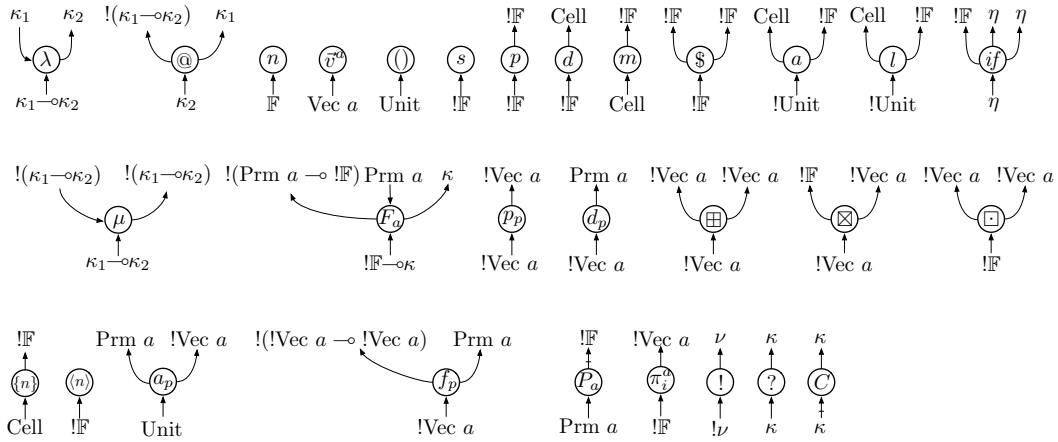


Figure 5.4: Connection of edges

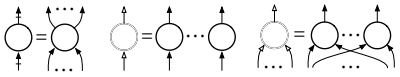
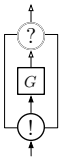


Figure 5.5: Conventions for multiple nodes and edges

not correspond to a syntactic construct but are created at runtime to represent dataflow graphs: $\{n\}$-node (memory cell), P_a-node (parameters) and π_i^a-nodes (projection). Finally, the !-node, ?-node, and C-node (contraction) are markers used to guide sub-graph copying. The connection of edges via nodes must satisfy the typings in Fig. 5.4 where $\vec{v}^a$ is a vector.

We also define a special collection of nodes in the form of !-boxes as follows:



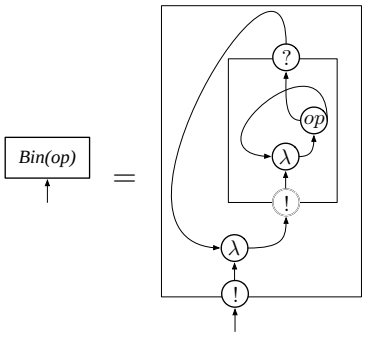
where every !-nodes, called the primary door, are coupled with a set of ?-nodes, called the auxiliary doors, and every node between them are inside the box. These boxes help with copying sub-graphs, which will be shown when we introduce rewrite transitions.

$$[\![Unit]\!] = !Unit \qquad [\![\mathbb{F}]\!] = !\mathbb{F} \qquad [\![Cell]\!] = Cell$$
$$[\![Vec\ a]\!] = !Vec\ a \qquad [\![Prm\ a]\!] = Prm\ a \qquad [\![\tau \to \tau']\!] = !([\![\tau]\!] \multimap [\![\tau']\!])$$

Figure 5.6: Translation of TSD types

When drawing graphs, certain diagrammatic conventions are employed. Ports are not represented explicitly, and their labels are only shown when they are not obvious. An edge with a bar at its tail represents zero or more ports, while a double-stroke edge represents a bunch of the same nodes and ports in parallel (Fig. 5.5).

We now define the translation of a simTSD term to graphs in Fig. 5.6, Fig. 5.7 and Fig. 5.8 respectively. In below, we will show an example of the translation, but first we need to define the below shorthand for binary primitive operations:



Notice that all the primitive operations are eta-expanded to support partial application.

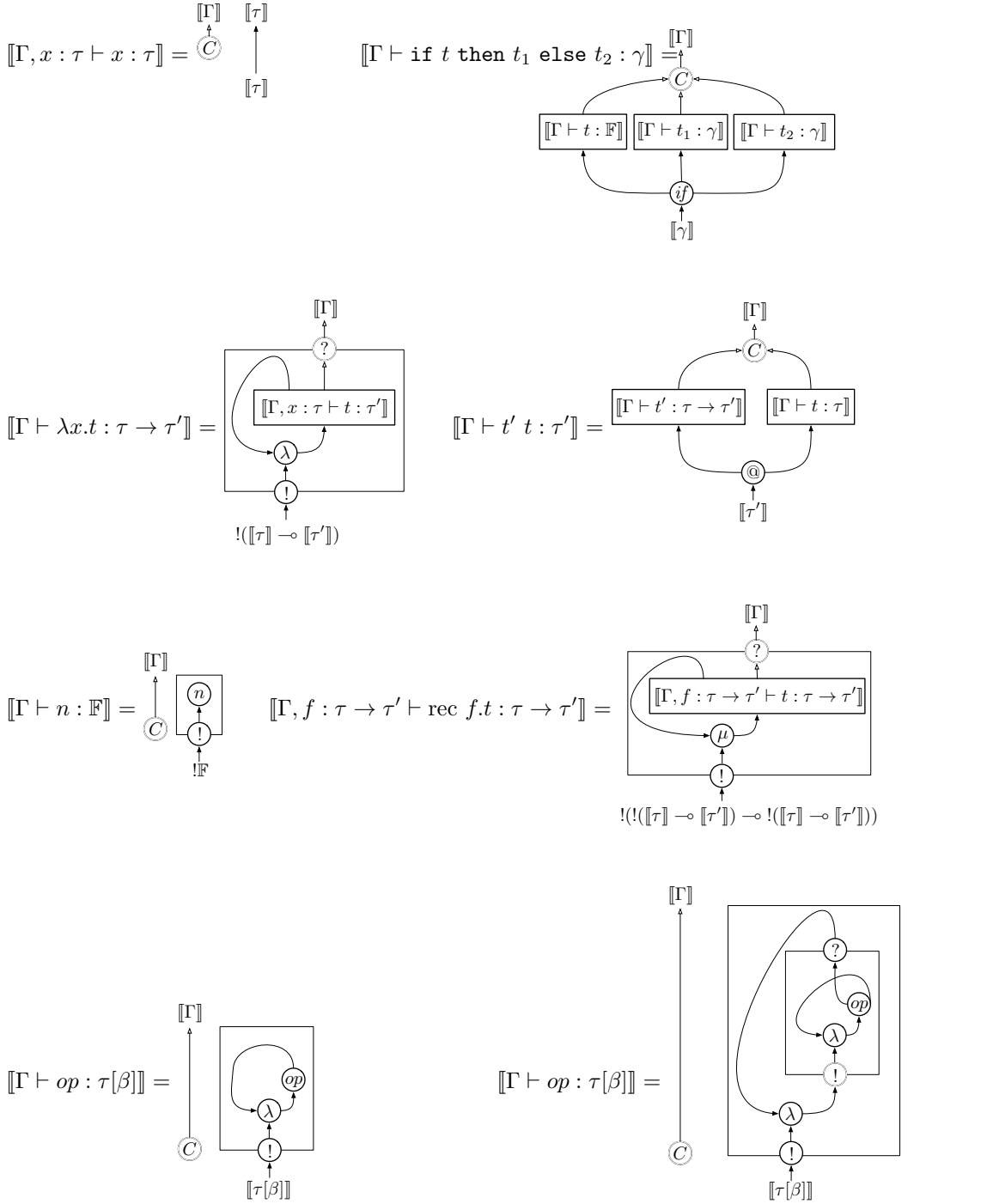
The translation of term $(\lambda x.x + x)(1 + 2)$ is shown in Fig. 5.9. Notice that there is an unreachable C-node hanging above the graph `Bin(+)` on the left. It is because the typing rule of application requires both sides to have the same typing context. For more examples, the readers are referred to the online visualiser[1] which not only shows the translation but also the evaluation of a TSD term.

5.2.2 Machine states

As mentioned in the informal semantics before, the machine execution is given by a token traversing the graph while collecting data and guiding it to rewrite. We now formally define the token data in Def. 5.2.1 and machine states in Def. 5.2.3.

Definition 5.2.1 (Evaluation tokens)**.** For any graph G, the state of an evaluation token (e, d, f, S, B) consists of a *port* $e \in Port_G$ indicating the token position, a *direction* d, a

[1] `https://cwtsteven.github.io/TSD-visual/`



where op is a unary operation.

where op is a binary operation.

Figure 5.7: Translation of TSD terms (1)

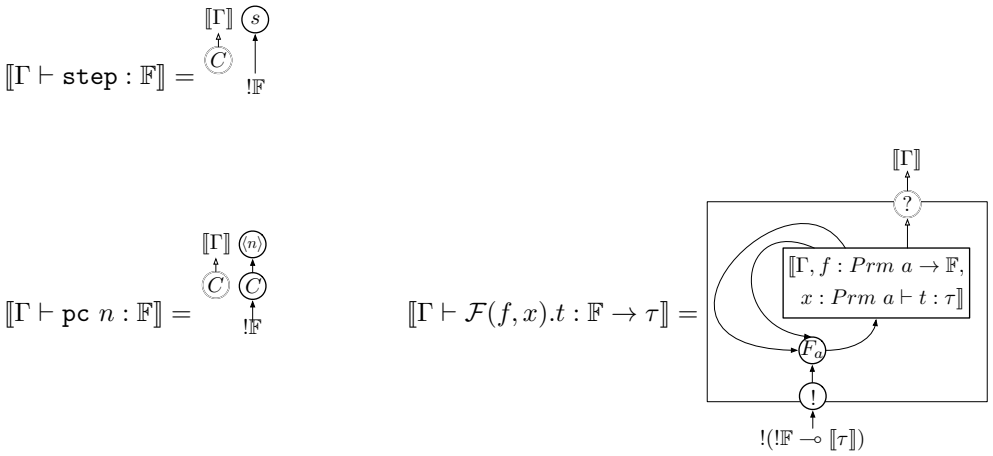


Figure 5.8: Translation of TSD terms (2)

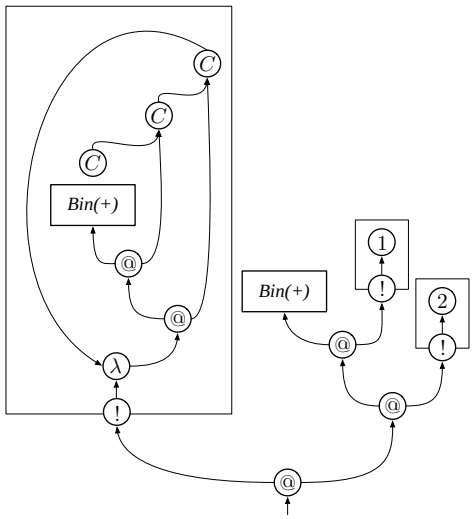


Figure 5.9: Translation of $(\lambda x.x + x)(1 + 2)$

rewrite flag f, a *computation stack* S and a *box stack* B with

$$
\begin{aligned}
d &::= \uparrow \mid \downarrow \\
f &::= \square \mid @ \mid \texttt{if} \mid ! \mid \mu \mid \$ \mid m \mid p \mid l(i) \mid a(n,i) \mid a(\vec{v}^b, i) \mid r(i) \mid sp \mid s \mid f(n) \\
&\quad \mid \boxplus \mid \boxtimes \mid \boxdot \\
S &::= \square \mid \star : S \mid (\lambda, -) : S \mid (n, -) : S \mid (n, g) : S \mid (n, i) : S \mid ((), -) : S \\
&\quad \mid \texttt{if}_0 : S \mid \texttt{if}_1 : S \mid (\vec{v}^b, -) : S \mid (\vec{v}^b, g) : S \mid (\vec{v}^b, i) : S \\
B &::= \square \mid i : B
\end{aligned}
$$

where $i \in Port_G$ and $b \in Name_G$.

In the definition above, e denotes the current *position* of the token. The *rewrite flag* indicates that whether a graph rewrite is ready to be made, with $\square$ meaning that no rewrites can apply. These flag will be raised during execution after certain transitions, which will be introduced soon in this section. Every flag correspond to the respective nodes in the graph except for *sp*, which is the transition of switching from evaluation-mode to propagation-mode. The *computation stack* tracks the intermediate results of the evaluation, while the *box stack* tracks duplication of values. Together the two stacks guide the transition of token. Intuitively, the token asks the value of a graph by having $\star$ on the top of the computation stack. After traversing the graph it will return either $(\lambda, -)$, (n, x), $(\vec{v}^b, x)$ or $((), -)$ as the result. The first part (λ,n,$\vec{v}^b$,()) is the return value while the second part indicates whether it is a cell or parameters ($i \in Port_G$ with i denoting the in-port of the cell), a graph (g) or a normal value ($-$). Notice that every vector values $\vec{v}^b$ are associated with a name b in G which indicates the family of vectors they belong to.

Definition 5.2.2 (Free/bound names)**.** A name $a \in \mathcal{Z}$ in a graph G is said to be *free* if there exists a P_a-node; or *bound* if there exists an F_a-node. We also define the set of *free* names in G as $FreeName_G$.

Definition 5.2.3 (Machine states)**.** A machine state (G, ζ, δ, P) is a quadruple consisting of

a *graph* G, a size mapping $\zeta : \mathit{FreeName}_G \to \mathbb{N}$ that maps any free names in G to a natural number indicating a vector space, an *eval token* δ (hereinafter referred to as the main token) and a *set of eval tokens* P (hereinafter referred to as the set of prop tokens).

The size mapping denotes the vector size of different vector families. Whenever a fusion binder is executed, the size mapping will create a new entry for the newly collected parameters. This will be discussed in the next section when transition rules are introduced. It is also essential when we prove that the language is symmetric in section 6.9. The main token is used during the evaluation mode to perform rewrites and to set up dataflow graphs. While the prop-tokens are only used during propagation mode in order to re-evaluate cells. As we will discuss below, the set of transitions used by prop-tokens is strictly a subset of that of the main token.

We also define two special machine states: initial and final states, denoting the starting and ending of an execution.

Definition 5.2.4 (Initial and final states)**.** For any graph $G : \Lambda \to \Gamma$, any $e \in \mathit{dom}(\Lambda)$ and any element X in the computation stack, we define the *initial* states and *final* states as $\mathit{Init}(G, \zeta, e) = (G, \zeta, e, \uparrow, \square, \star : \square, \square, \emptyset)$ and $\mathit{Final}(G, \zeta, e, X) = (G, \zeta, e, \downarrow, \square, X : \square, \square, \emptyset)$ respectively.

5.2.3 Transitions

We define a binary relation on machine states called transitions $(G, \zeta, \delta, P) \mapsto (G', \zeta', \delta', P')$. Notice that sometimes we represent the state not as a quadruple, meaning that the components that are not shown remain unchanged. We divide the transitions into three categories: *dataflow transitions*, *evaluation transitions*, and *propagation transitions*. Dataflow transitions can be seen as special evaluation transitions that only apply to dataflow graphs. Together with the evaluation transitions, they govern how the main token interacts with the graph. Propagation transitions govern how prop-tokens move in the graph in which a prop-token is

ensured to be running on a dataflow graph.

As an overview, the DGoI machine executes terms by encoding them into computation graphs according to the definitions in Fig. 5.7, where the translation of a well-typed term, is denoted by $[\![\Gamma \vdash t : \tau]\!]$. An *execution* is any sequence of transitions starting from the initial state. An execution is said to *terminate* if the execution ends with the final state. The formal definition of a terminating graph is in Def. 5.2.5. Note that it is in a more general form where it quantifies over any size mapping ζ for the purpose of proofs in chapter 6. A normal execution of a closed term always start with $\zeta = \emptyset$. An execution is said to be *safe* if it terminates or at any state there is an applicable rule. They will be formally defined in section 6.8.

Definition 5.2.5 (Terminating graphs)**.** For any graph $G : \Lambda \to \emptyset$ and any input $e : \kappa \in dom(\Lambda)$, $(G, \zeta, e) \Downarrow$ iff there exists some G' such that $Init(G, \zeta, e) \mapsto^* Final(G', \zeta', e, X)$ for some X in the computation stack and ζ'.

During an execution, when the main token hits an s-node (`step`), the machine will enter the propagation mode where a prop-token will be spawned at the root of each cell. Propagation transitions will then be carried out, and the main token is suspended. After all prop-tokens finish their evaluation, cells are updated, and the main token resumes and the machine goes back to evaluation mode. This mode-switching behaviour is captured by Def. 5.2.8. But first we need to define the set of initial and final prop-tokens which corresponds to the starting and the ending state of propagation mode.

Definition 5.2.6 (Initial prop-tokens)**.** We define the set of initial prop tokens $Init^P$ as

$$\{(e_0, \uparrow, \square, \star : \square, \square), ..., (e_n, \uparrow, \square, \star : \square, \square)\}$$

where $\{e_0, ..., e_n\}$ is the set of outputs of all $\{n\}$-nodes in the graph.

Definition 5.2.7 (Final prop-tokens)**.** We define the set of final prop tokens $Final^P(\vec{X})$ as

$$\{(e_0, \downarrow, \square, X_0 : \square, \square), ..., (e_n, \downarrow, \square, X_n : \square, \square)\}$$

where $\{e_0, ..., e_n\}$ is the set of outputs of all $\{n\}$-nodes in the graph, $\vec{X} = \{X_0, ..., X_n\}$ and $X_0, ..., X_n$ are elements of the computation stack.

Definition 5.2.8 (Mode-switching transitions)**.** We define the following special transitions:

1. $(G, \zeta, (e, \uparrow, \square, \star : S, B), \emptyset) \mapsto (G, \zeta, (e, \uparrow, sp, \star : S, B), Init^P)$ triggering the propagation when e is the incoming port of an s-node , and
2. $(G, \zeta, (e, \uparrow, sp, \star : S, B), Final^P(\vec{X})) \mapsto (G', \zeta, (e, \downarrow, s, (b, -) : S, B), \emptyset)$ terminating the propagation as follows: for any $\{n\}$-node in G that are not inside any boxes, let the return value of the eval token sitting on its output be (n', x), then if $n \neq n'$, the $\{n\}$-node is updated to $\{n'\}$ in G'. The value of b is 1 if there are updates and 0 otherwise where $1, 0 \in \mathbb{F}$.

Now we define the three categories of transitions in details.

Dataflow transitions

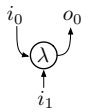
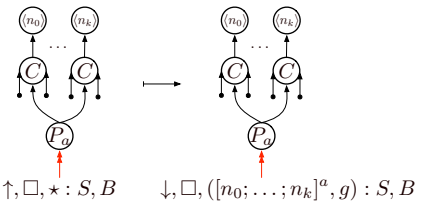


Figure 5.10: Port labels

The transition rules for dataflow graphs are given in Fig. 5.11. The token is at a distinguished port in the graph, and the node column represents the label of one of the two nodes of the port. The second column ('e') represents whether it is an incoming (i) or an outgoing (o) port. We labelled the ports by its left-to-right order from zero as in Fig. 5.10. The next three columns represent the state of the direction d, flag f, and computation stack S. The next column represents the new port e visited by the token, relative to the node. The final three columns represent the new values of the direction, flag, and computation stack.

Node $n, \vec{v}^a$**:** If the token is at the first incoming port i_1 of these nodes travelling along the direction of the edge $d = \uparrow$ the token will reverse directions $d = \downarrow$ and the top of the stack will be replaced by information that a n-node or a $\vec{v}^a$-node has been encountered by

node	e	d	f	S	$\mapsto$	e	d	f	S
n	i_0	$\uparrow$	$\square$	$\star$:S		i_0	$\downarrow$	$\square$	$(n, -)$:S
$\vec{v}^a$	i_0	$\uparrow$	$\square$	$\star$:S		i_0	$\downarrow$	$\square$	$(\vec{v}^a, -)$:S
d	i_0	$\uparrow$	$\square$	S		o_0	$\uparrow$	$\square$	S
d	o_0	$\downarrow$	$\square$	(n, i):S		i_0	$\downarrow$	$\square$	(n, g):S
d_p	i_0	$\uparrow$	$\square$	S		o_0	$\uparrow$	$\square$	S
d_p	o_0	$\downarrow$	$\square$	$(\vec{v}^a, i)$:S		i_0	$\downarrow$	$\square$	$(\vec{v}^a, g)$:S
For $\% \in \$ \cup \boxplus, \boxtimes, \odot$ (binary operations):									
$\%$	i_0	$\uparrow$	$\square$	S		o_1	$\uparrow$	$\square$	S
$\%$	o_1	$\downarrow$	$\square$	S		o_0	$\uparrow$	$\square$	$\star$:S
$\%$	o_0	$\downarrow$	$\square$	$(p, x) : (q, y)$:S		i_0	$\downarrow$	$\%$	$(\%\ p\ q, g)$:S
where $x \neq -$ or $y \neq -$									
$\%$	i_0	$\downarrow$	$\%$	$(\%\ a\ b, g)$:S		i_0	$\downarrow$	$\square$	$(\%\ a\ b, g)$:S
if	i_0	$\uparrow$	$\square$	S		o_0	$\uparrow$	$\square$	S
if	o_0	$\downarrow$	$\square$	$(0, g)$:S		o_1	$\uparrow$	$\square$	$\star : \texttt{if}_0$:S
if	o_0	$\downarrow$	$\square$	$(n \neq 0, g)$:S		o_1	$\uparrow$	$\square$	$\star : \texttt{if}_1$:S
if	o_1	$\downarrow$	$\square$	S		o_2	$\uparrow$	$\square$	$\star$:S
if	o_2	$\downarrow$	$\square$	$(p, x) : Y : \texttt{if}_0$:S		i_0	$\downarrow$	$\square$	(p, g):S
if	o_2	$\downarrow$	$\square$	$X : (q, y) : \texttt{if}_1$:S		i_0	$\downarrow$	$\square$	(q, g):S
!	i_0	$\uparrow$	$\square$	S		o_0	$\uparrow$	!	S
!	o_0	$\downarrow$	$\square$	S		i_0	$\downarrow$	$\square$	S
!	o_0	$\uparrow$	!	S		o_0	$\uparrow$	$\square$	S
if the !-node is not connected by a C-node									
$\{n\}$	i_0	$\uparrow$	$\square$	$\star$:S		i_0	$\downarrow$	$\square$	(n, i_0):S
$\langle n \rangle$	i_0	$\uparrow$	$\square$	$\star$:S		i_0	$\downarrow$	$\square$	(n, g):S
π_i^a	i_0	$\uparrow$	$\square$	S		o_0	$\uparrow$	$\square$	S
π_i^a	o_0	$\downarrow$	$\square$	$(\vec{v}^a, x)$:S		i_0	$\downarrow$	$\square$	$(\vec{v}^a.(i), g)$:S
where $\vec{v}^a.(i)$ is the i-th element of $\vec{v}$									



node	e	d	f	S	B	$\mapsto$	e	d	f	S	B
C	i_k	$\uparrow$	$\square$	S	B		o_0	$\uparrow$	$\square$	S	i_k:B
C	o_0	$\downarrow$	$\square$	S	i_k:B		i_k	$\downarrow$	$\square$	S	B

Figure 5.11: Dataflow transitions

placing $(n, -)$ or $\vec{v}^a$ at the top. For reasons of well-formedness, a $\star$ must have been at the top of S. This rule reflects the fact that a constant-term is a value which requires no further evaluation.

Node d, d_p: They represent the dereferencing operator, where incoming-port tokens are propagated to the outgoing port(s). When they come back along the outgoing port, they change the value on the top of the stack from (v, i) to (v, g) indicating that this value is now considered a graph instead of a cell/parameters.

Arithmetic and vector operations: If the token reaches the sole incoming port i_0 of these nodes, it will pass through and move to its first outgoing port o_1 (right out-port), reflecting the right-to-left call-by-value evaluation. As it returns $d = \downarrow$ along the same port o_1 it is passed to the zeroth outgoing port o_0 (left out-port) with changed direction $d = \uparrow$ and add a $\star$ to the top of the stack. Finally, as it returns $d = \downarrow$ on the same port it is passed to the incoming port i_0 with $d = \downarrow$, with the values of the branches m, n from the top of the stack and replace them with the value of the operator as applied to them \$ m n. The tags $x, y \in \{-, g\}$ but at least one should be g. This is because we want to mix constant nodes into the dataflow graph, and they result in a $-$ tag, whereas graph result in a g tag. If both of them are $-$ it means that there is a sub-graph made just of constants, which would be part of the evaluation transitions.

Node if: The 0-th outgoing edge connects to the condition graph, the 1-th edge to the *then*-branch and the 2-nd edge to the *else*-branch. When the condition is a dataflow graph, the if-node then behaves like a ternary operator, i.e. both branches will be evaluated. After returning from the 0-th outgoing edge, the top of the stack is inspected and a corresponding *flag* is added to the stack, with if_0 denoting that the result from *else*-branch should be taken and if_1 denoting the opposite. Either way, the *then*-branch will be evaluated first and then *else*-branch. Therefore, as seen in the final two rules, there are two values before the flag where X, Y represent any element of the stack. If the flag is if_0, the first value is taken as it represents the result from the *else*-branch and vice

versa. The usual behaviour of *if* (picking only one branch) is part of the evaluation transitions and will be discussed soon.

Node !: This node, also found in conventional GOI abstract machines, indicates that a box is about to be processed. The rewrite rule about to be triggered is indicated by the flag becoming $f = !$. For example, if the !-node is connected by a C-node under it, a rewrite will be triggered to copy the !-box in question. If not, then the ! flag is simply changed back to $\square$ (as shown in the last rule of !) and the execution continues.

Node $\{\underline{n}\}$: The cell storing value n is dealt with almost as a standard integer value. The only difference is that the port i_0 is also placed on top of S along with the value n, indicating it is a cell indeed and to be possibly used by datflow graph operations, e.g. assignment.

Node π_i^a: This is similar to d-node except that when the token comes back from the out-port to the in-port, it extracts the ith-element from the vector and replaces the top of the stack with it.

Node P_a: This node is similar to a constant, where the token simply reverses its direction and replaces $\star$ with the value of the parameters on the stack. This transition assumes the P_a node is always in the form shown in the table. Notice that an arrow with a block dot at its end indicates that it is connected by some graph that is irrelevant to this transition.

Node C: Contraction nodes use the other stack, the box-stack, to remember along which of the several incoming ports a token arrived so that it can return to the same port.

The semi-strictness of the if-node (becoming a ternary operator whenever the condition is a dataflow graph) is required for soundness because of `step` propagation. Consider the code[2]:

$$\textbf{let } x = \text{ref } 1 \textbf{ in } \text{ref}(\text{if } x < 2 \text{ then } f\ 3 \text{ else } f\ 4)$$

Suppose the else-branch, i.e. $f\ 4$, is not evaluated beforehand, if x is changed to 2 then

[2] `https://cwtsteven.github.io/TSD-visual?ex=if`

$f\ 4$ would need to be evaluated during propagation mode. If $f\ 4$ is impure, it will possibly lead to race condition where different prop-tokens trying to access the same part of the graph and to perform rewrites such as copying. The rule however is inefficient because whenever x changes, both of the branches will need to be re-evaluated, which is unnecessary. An optimised version of the rule which only evaluates one branch could be written, but it is more complicated. Moreover, the inefficiencies introduced here do not change the asymptotic complexity of graph evaluation, as will be seen shortly.

The dataflow transitions are enough for evaluating dataflow graphs, and it will be shown later when we discuss the soundness proof in section 6.2.

Evaluation transitions

In addition to dataflow transitions, we define a set of evaluation transitions which combine with dataflow transitions gives the ability to evaluate a program. We further separate these transitions into *pass* and *rewrite* rules. Pass rules are responsible for searching the next redex and setting up the scene for rewrites to happen, while rewrite rules are the actual rules that perform reduction. The pass rules are given in Fig. 5.12.

Node λ: If the token is at the first incoming port i_1 of a λ node travelling along the direction of the edge $d = \uparrow$ the token will reverse directions $d = \downarrow$ and the top of the stack will be replaced by information that a λ node has been encountered by placing $(\lambda, -)$ at the top. For reasons of well-formedness, a $\star$ must have been at that op of S. This rule reflects the fact that a λ-term is a value which requires no further evaluation.

Node @: If the token reaches the sole incoming port i_0 of an application node it will pass through and move to its first outgoing port o_1 (right out-port), reflecting the right-to-left call-by-value evaluation. As it returns $d = \downarrow$ along the same port o_1 it is passed to the zeroth outgoing port o_0 (left out-port) with changed direction $d = \uparrow$ and the top of the stack become $\star$. Finally, as it returns $d = \downarrow$ on the same port it is passed to the incoming port i_0 with $d = \downarrow$ in preparation for the rewrite corresponding to application. Note that

node	e	d	f	S	$\mapsto$	e	d	f	S
λ	i_1	$\uparrow$	$\square$	$\star$:S		i_1	$\downarrow$	$\square$	$(\lambda, -)$:S
@	i_0	$\uparrow$	$\square$	S		o_1	$\uparrow$	$\square$	S
@	o_1	$\downarrow$	$\square$	S		o_0	$\uparrow$	$\square$	$\star$:S
@	o_0	$\downarrow$	$\square$	$(\lambda, -) : X$:S		i_0	$\uparrow$	@	$\star$:S
()	i_0	$\uparrow$	$\square$	$\star$:S		i_0	$\downarrow$	$\square$	$((), -)$:S
For $\omega \in p, r, m, p_p$ (unary operations):									
ω	i_0	$\uparrow$	$\square$	S		o_0	$\uparrow$	$\square$	S
p	o_0	$\downarrow$	$\square$	(n, x):S		i_0	$\downarrow$	p	$(n, -)$:S
p_p	o_0	$\downarrow$	$\square$	$(\vec{v}^a, x)$:S		i_0	$\downarrow$	p	$(\vec{v}^a, -)$:S
r	o_0	$\downarrow$	$\square$	(n, i):S		i_0	$\uparrow$	$r(i)$	$\star$:S
m	o_0	$\downarrow$	$\square$	(n, x):S		i_0	$\downarrow$	m	$(n, -)$:S
For $\# \in \{a, l, a_p, f_p\}$ (binary operations):									
#	i_0	$\uparrow$	$\square$	S		o_1	$\uparrow$	$\square$	S
#	o_1	$\downarrow$	$\square$	S		o_0	$\uparrow$	$\square$	$\star$:S
For $\% \in \$ \cup \{\boxplus, \boxtimes, \odot\}$:									
%	o_0	$\downarrow$	$\square$	$(p, -) : (q, -)$:S		i_0	$\downarrow$	%	$(\%\ p\ q, -)$:S
a	o_0	$\downarrow$	$\square$	$(m, i) : (n, x)$:S		i_0	$\downarrow$	$a(n, i)$	$((), -)$:S
a_p	o_0	$\downarrow$	$\square$	$(\vec{v}^a, i) : (\vec{u}^a, x)$:S		i_0	$\downarrow$	$a(\vec{u}^a, i)$	$((), -)$:S
l	o_0	$\downarrow$	$\square$	$(m, i) : (n, x)$:S		i_0	$\downarrow$	$l(i)$	$((), -)$:S
f_p	o_0	$\downarrow$	$\square$	$(\lambda, -) : (\vec{v}^a, y)$:S		i_0	$\uparrow$	$f(k)$	$\star$:S
where $x \in \{-, g\}$ and k is the length of $\vec{v}$									
if	o_0	$\downarrow$	$\square$	$(0, -)$:S		o_1	$\uparrow$	*if*	$\star$:S
if	o_0	$\downarrow$	$\square$	$(n \neq 0, -)$:S		o_2	$\uparrow$	*if*	$\star$:S
μ	i_1	$\uparrow$	$\square$	S		i_1	$\uparrow$	μ	S
F_a	i_1	$\uparrow$	$\square$	$\star$:S		i_1	$\downarrow$	$\square$	$(\lambda, -)$:S

Figure 5.12: Pass rules of evaluation transitions

it is expected that the last value encountered on the left side was a lambda term, since the rule expects $(\lambda, -)$ at the top, and the second element X being the argument. The imminence of the rewrite rule is indicated by the rewrite flag $f = @$ shown in Fig. 5.13.

Nodes (): This node represents the unit value, so its behaviour is quite similar to that of the λ node, reflecting the token and placing the value on S.

Operation nodes These nodes behave similarly to @. Incoming-port tokens are propagated to the outgoing port(s). When they come back along the outgoing port, the behaviour is different according to whether they are unary or binary operators. If they are unary, the token is passed to the incoming port and the rewrite flag f is used to indicate an imminent rewrite. If they are binary, then the zeroth outgoing port is visited first.

Node *if*: Depending on the previous pass rule, the token will either be sent to the first or the second branch and the rewrite flag will be changed to *if* to indicate a possible rewrite.

Node μ: The recursion node immediately prepares the machine for a rewrite by changing the flag to $f = \mu$.

Node F_a: This is similar to the case of node λ where the top of the stack is changed from $\star$ to $(\lambda, -)$ to indicate there is no further evaluation for values.

Note that certain nodes, such as ?, have no associated rules because the token is never supposed to reach them.

The rewrite rules are given in Fig. 5.13, Fig. 5.14, Fig. 5.15 and Fig. 5.17.

The *Beta* rule: The essential computational rule of the lambda calculus is the removal of the abstraction-application pair. The 0-th ingoing edge of the λ-node is now re-wired to the 1-th outgoing port of the application node @, which is the argument. This is a 'small' beta law since it involves no copying of the argument. To anticipate a little, the copying will be performed by the contraction nodes.

The *s*-rule: It represents the 'step' command which evaluates the dataflow graph, which will be discussed later in section 5.2.3. This rewrite, with a token exiting the incoming port $d = \downarrow$ is performed after this evaluation. The dataflow evaluation leaves a value n

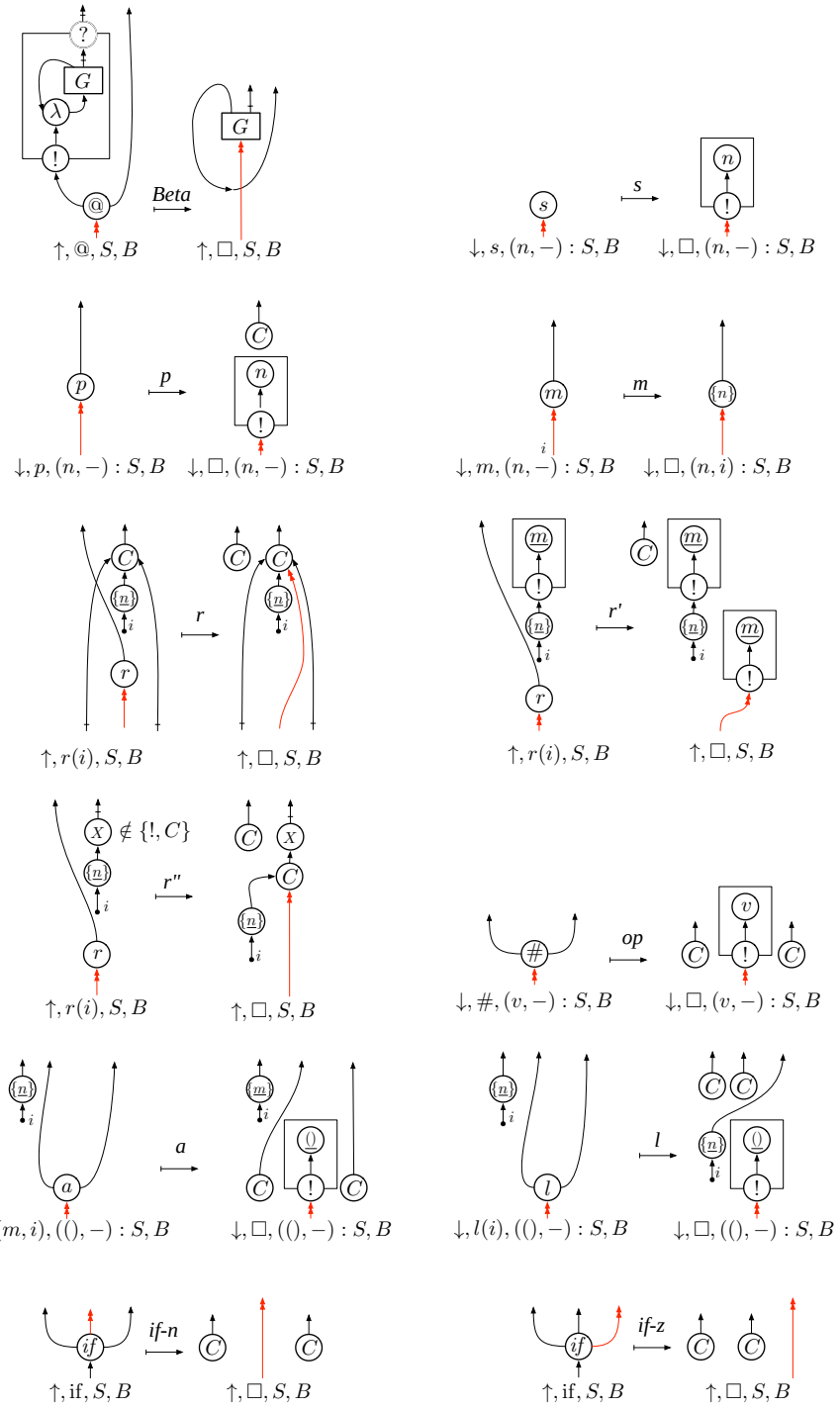


Figure 5.13: Computation rewrites

on S indicating that n nodes were updated, so the s-node is replaced by a constant n value box.

The p-rule: It represents the 'peek' command which reads the value off the dataflow graph, as pointed at by the p node, and creates a constant n valued box from it. Note that the outgoing port of the p node is replaced by a C-node without ingoing ports representing variable weakening, indicating that port became unreachable.

The m-rule: It represents cell creation, and is executed by replacing the m node by an n-initialised cell node. The value n is retrieved off the stack n, where it is augmented with the port i so that it can be used by other dataflow graph manipulating operations.

The r, r', r''-rules: It computes the dependency ('root') of a graph. There are three versions of the rule (r, r', r'') depending on whether the requested dependency is incoming to a contraction, constant, or some other node. In all cases, the token 'jumps' to the incoming port of the cell, labelled by i. Note that this rule is 'non-local', involving edge i located in some arbitrary point in the graph. The node which is at the nock of the arrow is irrelevant, and it is not shown.

The *op*-rules: compute operators by replacing the operation node with the resulting value as returned by the token and left on S. This is very similar to how stack machines generally evaluate operators. The outgoing ports of the operator are detached from the graph via weakening.

The a-rule: implements assignment. A port i is left at the top of S along with a computed integer m (in the form (m, i)). The result of the assignment operation is unit () but the value of the cell with incoming port x is updated to m.

The l-rule: implements linking. At the top of S there is a port x (tagged as $l(i)$). The operation returns (), like assignment, but the outgoing port of the cell pointed at by port i is reassigned to the second argument i of a.

The *if*-rules: Depending on the previous pass-rule, the token is either on the first or second branch with the rewrite flag being *if*. If it is the former case, then the second branch

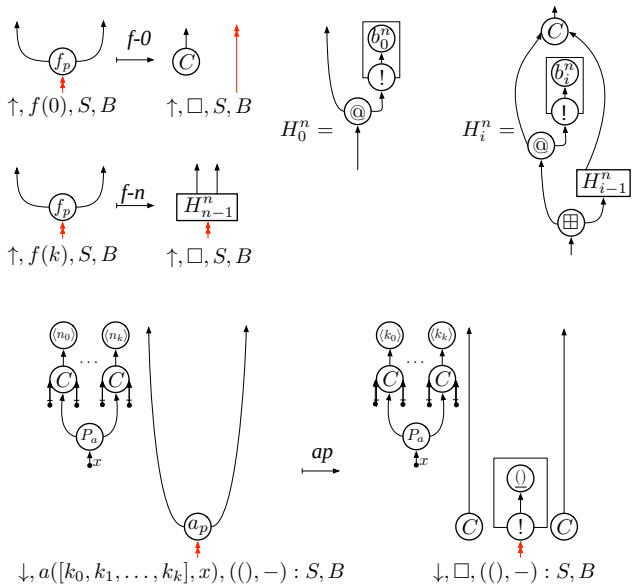


Figure 5.14: Computation rewrites for TSDextension

and the conditional branch will be rewired to a C-node indicating they are no longer going to be evaluated. In both cases, the rewrite flag will be changed back to $\square$.

The f-rules: It represents folding a function over the base vectors. If the flag is $f(0)$ it means that the length of the parameters in question is zero and there is nothing to fold on. Otherwise, this rule will generate the appropriate base vectors where b_i^n is an all-zeros vector with length n and its ith element being 1, e.g. if $k = 3$ then the base vectors will be $\{[1; 0; 0], [0; 1; 0], [0; 0; 1]\}$, applies the function (the sub-graph connected by o_0) to each of these base vectors and finally aggregate the results by creating a chain of $\boxplus$-nodes.

The ap-rules: This rule is analogous to the a-rule but assign the value of a vector parameters with a vector in the same name-family.

The Fuse-rule: In this rule, the graph U is the largest reachable graph from the o_1-port of the @-node that does not contain any P_b-nodes, provisional constants (in top level) and any cells ($\{n\}$-node) that depend on a constant. In order words, the graph H is basically an array of P_b-nodes because the form of a P_b-node is fixed, i.e. it connects to an array of C-nodes with each connecting to a provisional constant. Since the o_1-port of the

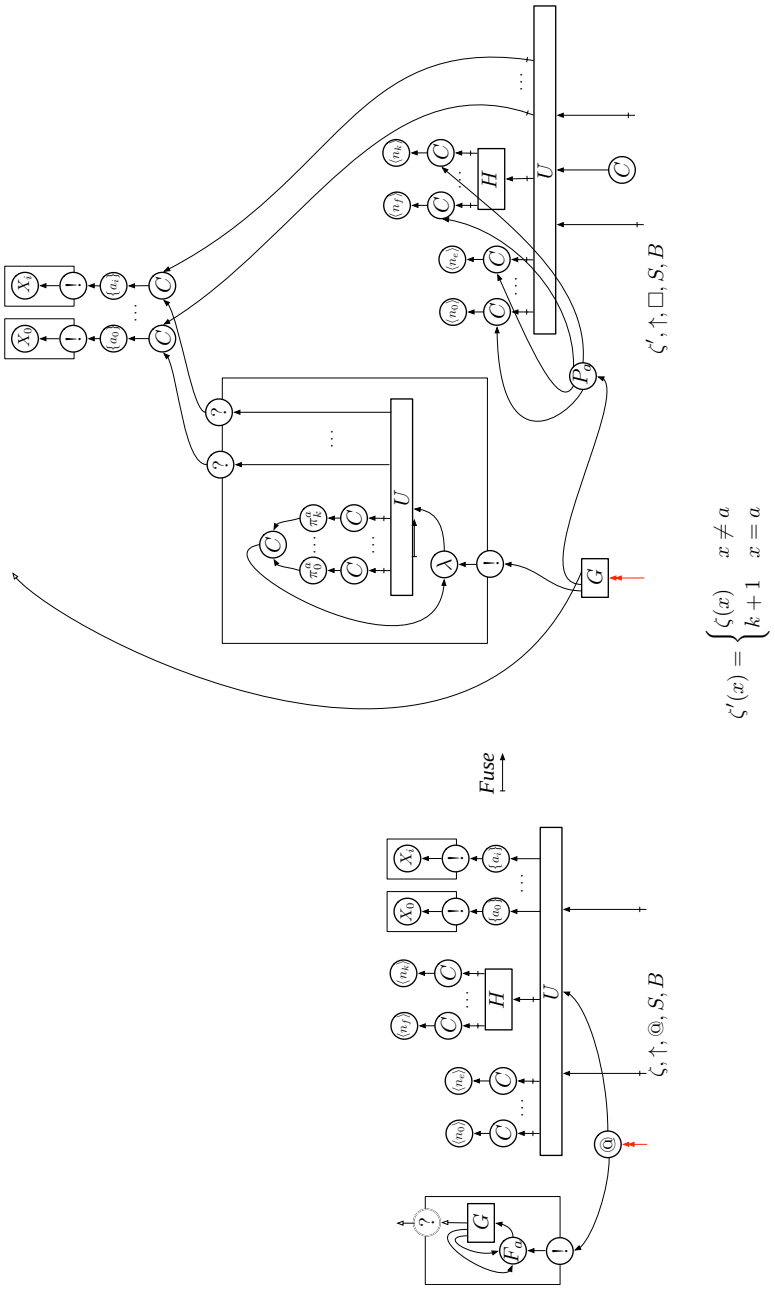


Figure 5.15: Fusion rewrite

@-node is of $\mathbb{F}$ type, it must be connecting to a dataflow graph. Therefore, U must be the 'skeleton' of a dataflow graph representing the reactions relative to the input cells and provisional constants. These provisional constants are collected by an algorithm independent of how the graph is drawn, e.g. depth-first traversal, such that the transition is deterministic. It then creates a P_a-node connecting to these provisional constants. Furthermore, it also creates a λ-box which encapsulates a copy of the 'skeleton' graph U with π_i^a-nodes replacing the original provisional constants representing the projection of the ith-element of a vector. As one can imagine, the function can be applied to different vectors and have different results after evaluation. Moreover, the function also points towards the input cells which preserves the data dependencies. This rule is also the only rule that modifies the size mapping ζ which adds the map $(a, k+1)$ where a is the name of the parameters and $k+1$ the size of the collected parameters.

This rule is similar to the *graph-abstraction* rule in ITF [62] with the major difference being that we need to consider having cells in the graph U. In that case do we create copies for these cells or do we create dependencies for them? We chose the former, for example the fusion transition of the code below is illustrated by Fig. 5.16.

$$\texttt{let } x = \texttt{ref } 1 \texttt{ in}$$

$$\texttt{let } y = \texttt{ref } (\texttt{deref } x + \texttt{pc}2) \texttt{ in}$$

$$(\mathcal{F}(f, ps).t)\ (\texttt{deref } y + \texttt{pc}3)$$

As one can see, the cell y ($\{3\}$) is copied into the skeleton function. The problem with the latter approach is that the provisional constant `pc` 2 will not be captured as part of the parameters and there will not be a way to tune it afterwards.

The *Cc* rule: It combines two consecutive m-ary and n-ary contraction nodes in a single $m+n$-ary contraction node.

The *C-!* rule: It removes redundant 'unary' contractions.

The δ rule: It is the main rule of the contraction node, and it leads to the copying of the

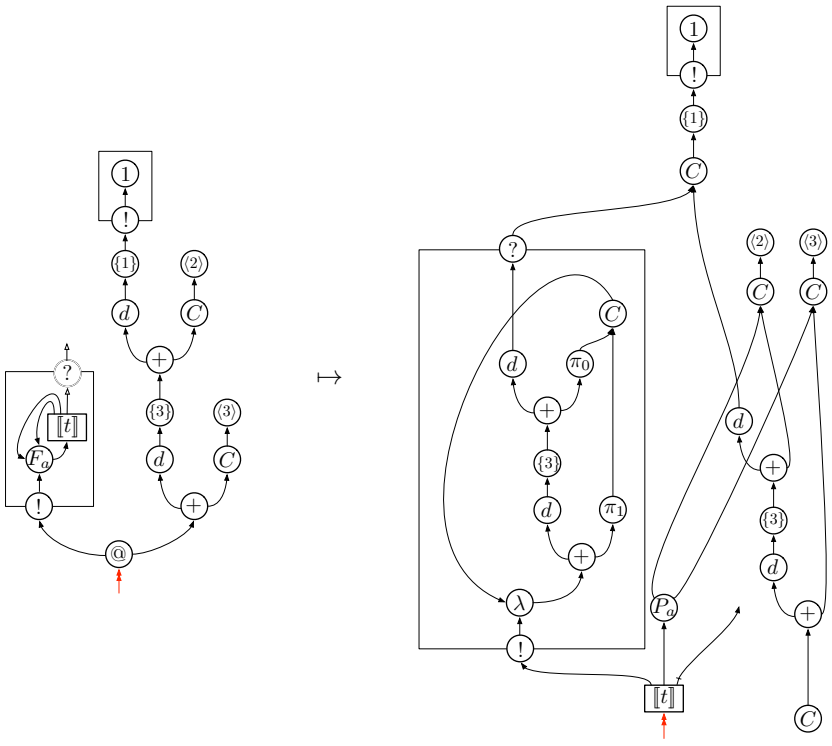


Figure 5.16: Example of fuse-rule

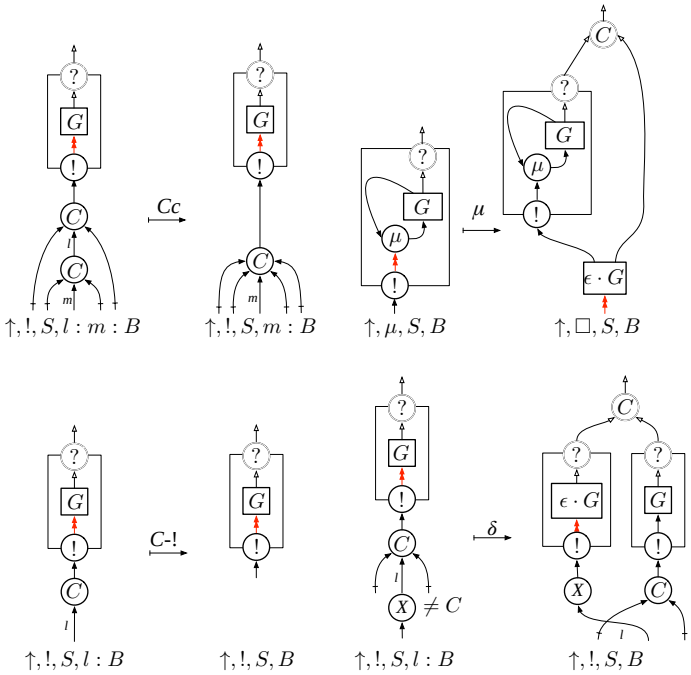


Figure 5.17: Copying rewrites

box just above it. Note that this is the main purpose of boxes, to delimit copyable sub-graphs for copying by the contraction node. In both this rule and the μ-rule, $\epsilon \cdot G$ is an action that replaces every *bound* name in G with a fresh name according to the permutation ϵ.

The μ-rule: It is not a contraction rule, but it is very similar. It represents the 'unfolding' of a recursive call by copying the body G.

Propagation transitions

Propagation transitions are the transitions performed by prop-tokens. Because prop-tokens can only evaluate dataflow graphs, they basically follow the dataflow transition rules (which is also why we separate dataflow transitions from evaluation transitions).

Definition 5.2.9 (Propagation transitions)**.** A *propagation* transition is any transition in the form of $(G, \delta, \{p\} \cup P) \mapsto (G, \delta, \{p'\} \cup P)$ where $(G, \delta, p, \emptyset) \mapsto (G, \delta, p', \emptyset)$ is a dataflow transition.

Chapter 6

Theoretical Results

There are five major results obtained for the language simTSD: *determinism*, *type soundness*, *symmetry*, *graph equivalences* and *efficiency*. We will first discuss these theorems informally as an overview and show their proofs in details in the following sections.

Determinism (section 6.5) There are two sources of non-determinism in the transition system: (1) fresh names generated by δ-rewrite and the μ-rewrite; and (2) choosing which prop-tokens to proceed during propagation.

The former is straightforward. Since names do not actually affect the execution of a graph, and we can prove that given any permutation of names acting on the graph will not alter the behaviour. As for the latter, since prop-tokens are created during propagation on the out-port of each cell, their execution sequences must only contain dataflow transitions because cells are connected to a dataflow graph at all time. Because there is no rewrite in dataflow graphs, therefore there is no race condition between different prop tokens and thus the results converge and the overall behaviour of propagation is confluent. As a result, propagation can be seen as one single transition and thus the machine is deterministic.

Type soundness (section 6.8) The type soundness theorem is separated into two parts: (1) well-typed recursion free programs always terminate; and (2) well-typed programs do not get

stuck, which is formally defined in Des. 6.0.1.

Desideratum 6.0.1 (Type soundness)**.** *For any closed well-typed term* $- \vdash t : \tau$*, let* $G : \{r : [\![\tau]\!]\} \to \emptyset = [\![- \vdash t : \tau]\!]$ *then*

1. *if* t *is recursion-free then the execution from* $Init(G, \emptyset, r)$ *will terminate,*
2. *if* t *is with recursion then the execution from* $Init(G, \emptyset, r)$ *will not get stuck.*

Symmetry (section 6.9) By *symmetry* we mean that given a permutation of vector indices acting on a graph, the evaluation of the graph must respect the permutation as defined in Des. 6.0.2. It is easy to see that most of the operations such as vector addition are symmetric, however it is not the case with `foldp` because of effects. Consider the term `foldp` $(\lambda b.(\texttt{assignp}\ ps\ b); b)\ ps$ with $ps : Prm\ a$ and $(\lambda b.(\texttt{assignp}\ ps\ b); b) : Vec\ a \to Vec\ a$, since the semantics of `foldp` is to create a graph with a chain of $\boxplus$-node aggregating the result of the function applying to the set of base vectors, but the order of created base vectors are independent of the permutation, and thus it will result in a graph that does not respect the permutation. An example is shown in Fig. 6.1 where the graph G represents the function $\lambda b.\texttt{assignp}\ ps\ b; b$ and $ps = [2; 3]$. The upper parts of the figure show the evaluation of the graph representing the term `foldp` $(\lambda b.\texttt{assignp}\ ps\ b; b)\ ps$ whereas the lower parts show the evaluation of a graph that is permuted by a permutation ϕ which reverses the order of any vector on the upper graph. As shown in the figure, the f-2-rewrite creates the base vectors $[0; 1]^a$ and $[1; 0]^a$ in the same places for both graph. Since G is a function that will assign a vector to the P_a-node, and the evaluation of $\boxplus$ is right-to-left, after several steps, they will result in the same graph as shown in the figure. As a result, the final graph does not respect the permutation ϕ any more. Therefore, symmetry can only be achieved when the function passed to `foldp` and the context used by it is pure. Converting this requirement in terms of graph, it is to say the reachable graph of an f-node right before the rewrite happens has to be *pure*.

Desideratum 6.0.2 (Symmetry)**.** *For any closed graph* $G : \{e : \kappa\} \to \emptyset$ *and any permutation*

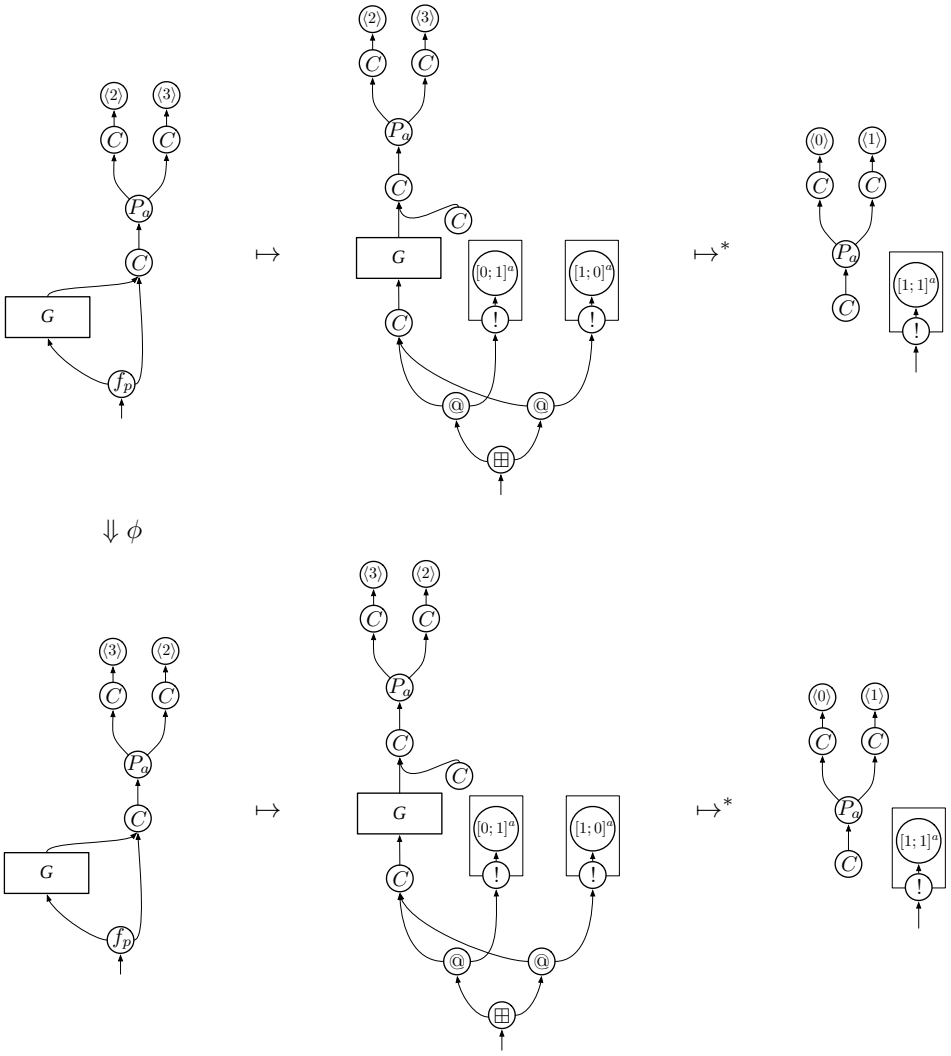


Figure 6.1: Example of `foldp`

on states ϕ, $Init(G, \emptyset, e) \Downarrow Final(G_1, e, \zeta_1, X)$ and $\phi \cdot Init(G, \emptyset, e) \Downarrow Final(G_2, e, \zeta_2, Y)$ and for any f-rewrite in the execution the reachable graph from the f-node is pure then $X = \phi \cdot Y$.

Graph equivalences (section 6.10) In particular, we prove the basic beta-equivalence in terms of graph. But first we need to introduce the notion of contextual equivalence graphically.

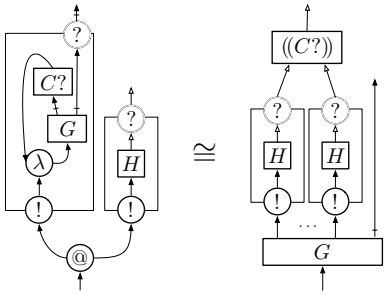
Definition 6.0.1 (Graph equivalence)**.** For any graph G_1, G_2 we say that $G_1 \equiv G_2$ if $Init(G_1, \emptyset, e) \Downarrow Final(G_1', e, \zeta_1, X)$ and $Init(G_2, \emptyset, e) \Downarrow Final(G_2', e, \zeta_2, Y)$ such that $X = Y$.

Definition 6.0.2 (Graph contexts)**.** A graph context $\mathcal{G}$ is an incomplete graph with a hole inside that can be plugged in with another graph H to form a complete graph $\mathcal{G}[H]$.

Definition 6.0.3 (Graphical contextual equivalence)**.** For any graph G_1, G_2 we say that $G_1 \cong G_2$ if for any context graph $\mathcal{G}$, $\mathcal{G}[G_1] \equiv \mathcal{G}[G_2]$.

Graphical beta-equivalence can be separated into three parts: (1) *rewiring* the connection between the bound variable and the argument (Sec. 6.10.2), (2) *copying* the box values representing individual substitution for each instance (Sec. 6.10.3) and (3) *engulfing* a box into another box representing substitution through other functions (Sec. 6.10.4). Together these three equivalences implies beta-equivalence, but they are also interesting in their own rights.

Desideratum 6.0.3 (Beta-equivalence)**.**



if H is not a μ-box

where $C?$ is a tree of  *and*  *; and $(\!(C?)\!)$ is a tree of*  *and*  *such that $C?$ and $(\!(C?)\!)$ have the same structure.*

The $C?$-graph represents all the instances of the variable used in the function, therefore substitution can be seen as giving a copy of the argument to every leaf in the graph $C?$.

Complexity of transitions (section 6.11) Assuming that a graph is created as a simple recursive data structure, then the following actions are atomic and only costs constant amount of time: token transitions from one place to another, copying a single node, creating a token and updating a cell value. Therefore, we can conclude that the time complexity of each transition is either constant or linear in the size of the graph.

Desideratum 6.0.4 (Complexity of transitions)**.** *Time complexity of all transitions are either constant or linear in the size of the graph.*

6.1 Conventions

In the following sections we will define certain graph compositions by the rules below, which are borrowed from string diagrams.

Definition 6.1.1 (Sequential graph composition)**.** For any graph $G : \Lambda \to (\Gamma \cup \Psi)$ and any graph $E : (\Gamma \cup \Pi) \to \Omega$, we define the sequential composition of the two graphs as $EG : (\Lambda \cup \Pi) \to (\Psi \cup \Omega)$.

Definition 6.1.2 (Parallel graph composition)**.** For any graph $G : \Lambda \to \Gamma$ and any graph $E : \Pi \to \Omega$, we define the parallel composition of the two graphs as $E \otimes G : (\Lambda \cup \Pi) \to (\Gamma \cup \Omega)$.

6.2 Dataflow graphs

In this section, we define a special form of machine graphs, namely dataflow graphs, which correspond to dataflow models in a program (Def. 6.2.3). However, before going into these definitions we first define some common properties of graphs.

Definition 6.2.1 (Paths)**.** A path is an ordered list of links $e_0 ... e_n$ such that if e_i is the in-port and e_{i+1} is the out-port of the same node.

Definition 6.2.2 (Reachability)**.** A node X is reachable from another node Y if there exists a path from any out-port of Y to any in-port of X.

Now we can define dataflow graphs as follows:

Definition 6.2.3 (Dataflow graphs)**.** A dataflow graph is a graph that consists of only the nodes in the following set

$$\{\$, \texttt{if}, C, d, n, \vec{v}^a, !, d_p, \boxplus, \boxtimes, \boxdot, P_a, \pi_i^a, \{n\}, \langle n\rangle\}$$

satisfying the following conditions:

1. for each node in the graph, either a) it is reachable from a $\{n\}$-node; or b) a $\{n\}$-node or a $\langle n\rangle$-node is reachable from it; or c) it is reachable from another node X such that either a $\{n\}$-node or a $\langle n\rangle$-node is reachable from X,
2. the output of any C-node is not connected to a !-box,
3. the left-output of any `if`-node is not connected to a !-box, and
4. at least one of the outputs of any \$-nodes, $\boxplus$-node, $\boxtimes$-node or $\boxdot$-node is not connected to a !-box.

Intuitively these graphs are formed only by constants, cells, parameters, provisional constants, their operations and the if-expression. The first condition ensures that a dataflow graph is defined around cells and provisional constants, for example a graph $(!n) + (!n)$ is not considered to be a dataflow graph but $\langle n\rangle + (!n)$ is. The other three conditions ensure that no rewrite rules can be applied to these graphs; as such, these graphs are said to be 'pure dataflow graphs'. In Sec. 6.5 we will show that this is the case. Since there cannot be any rewrite transitions when executing these graphs, it is easy to prove that they always terminate. This will also be proved in Sec. 6.5.

Apart from dataflow graphs, we also identify the following forms of graphs: *value dags* (Def. 6.2.4) and *term graphs* (Def. 6.2.5).

Definition 6.2.4 (Value dags)**.** A *value dag* is a directed acyclic graph of !-boxes and C-nodes.

Definition 6.2.5 (Term graphs)**.** A *term graph* is a graph that does not contain any $\{n\}$-nodes, $\langle n \rangle$-nodes nor any P_a-nodes.

Value dags are directed acyclic graphs consisting of only !-boxes and C-nodes. They correspond to the non-dataflow portion of the environment. In section 6.6 we will show that expanding all the contraction nodes in a value dag will result in a *value tree* that has the same observational behaviour. On the other hand, a term graph represents a term in a program that is yet to be evaluated.

6.3 Name equivalence

One of the sources of non-determinism comes from the δ-rewrite and the μ-rewrite where fresh names are introduced. However, this should not affect the behaviour of any program since picking different names do not change the shape of the graph. We first define a binary relation on machine states where one graph is a name-permuted copy of another, and later we show that this relation is a bi-simulation and thus an equivalence relation.

Definition 6.3.1. We define a binary relation on machine states $(G_1, \zeta_1, \delta_1, P_1) \overset{\alpha}{\sim} (G_2, \zeta_2, \delta_2, P_2)$ if there exists a bijection $\psi : \mathit{Name}_{G_1} \to \mathit{Name}_{G_2}$ such that $\psi \cdot G_1 = G_2$, $\psi \cdot \delta_1 = \delta_2$, $\psi \cdot P_1 = P_2$ and for any $a \in \mathit{Name}_{G_1}$ $\zeta_1(a) = \zeta_2(\psi(a))$.

Proposition 6.3.1. $\overset{\alpha}{\sim}$ *is a bi-simulation.*

Proof. We need to prove that for any machine states π_1, π_2 such that $\pi_1 \overset{\alpha}{\sim} \pi_2$ then (1) if $\pi_1 \mapsto \pi_1'$ then there exists a state π_2' such that $\pi_2 \mapsto \pi_2'$ and $\pi_1' \overset{\alpha}{\sim} \pi_2'$ and (2) if $\pi_2 \mapsto \pi_2'$ then there exists a state π_1' such that $\pi_1 \mapsto \pi_1'$ and $\pi_1' \overset{\alpha}{\sim} \pi_2'$.

Let $\pi_1 = (G_1, \zeta_1, \delta_1, P_1)$ and $\pi_2 = (G_2, \zeta_2, \delta_2, P_2)$ since $\pi_1 \overset{\alpha}{\sim} \pi_2$, therefore there exists a bijection $\psi : \mathit{Name}_{G_1} \to \mathit{Name}_{G_2}$ such that $\psi \cdot G_1 = G_2$, $\psi \cdot \delta_1 = \delta_2$, $\psi \cdot P_1 = P_2$ and for any

$a \in Name_{G_1}\ \zeta_1(a) = \zeta_2(\psi(a))$.

To prove (1) we need to consider all possible transitions $\pi_1 \mapsto \pi_1'$. Let $\pi_1' = (G_1', \zeta_1', \delta_1', P_1')$.

Evaluation transitions. For all the pass-rules since G_1, G_2 have the same shape we can apply the same transition to π_2 to get $\pi_2' = (G_2', \zeta_2', \delta_2', P_2')$. Pass rules do not create new names or modify any existing names therefore ψ remains unchanged and since the graph does not change at all we have $G_1' = G_1$, $G_2' = G_2$ and thus $G_2' = G_2 = \psi \cdot G_1 = \psi \cdot G_1'$. Also, because of that $\zeta_1' = \zeta_1$, $\zeta_2' = \zeta_2$ therefore for any $a \in Name_{G_1'}\ \zeta_1'(a) = \zeta_2'(\psi(a))$. Since in evaluation transitions $P_1 = P_2 = P_1' = P_2' = \emptyset$, we have $P_2' = \psi \cdot P_1'$. Also since all the pass rules respect the names, e.g. in the $\boxplus$-rule a vector under the same family is created, therefore we have $\psi \cdot \delta_1' = \delta_2'$.

For those rewrite-rules that do not create new names or modify existing names, they all respect the permutation ψ similar to the pass rules.

As for the δ-rewrite and μ-rewrite, fresh names are generated by some bijection ϵ_1 and ϵ_2. Therefore, there exists a bijection

$$\psi'(x) = \begin{cases} \psi(x) & \textit{if } x \in Name_{G_1} \\ \epsilon_2(\psi(y)) & \textit{if } x = \epsilon_1(y) \textit{ where } y \in Name_{G_1} \end{cases}$$

such that $\psi' \cdot G_1' = G_2'$, $\psi' \cdot \delta_1' = \delta_2'$, $\psi \cdot P_1' = P_2'$ and for any $a \in Name_{G_1'}\ \zeta_1'(a) = \zeta_2'(\psi(a))$.

Dataflow transitions. Dataflow transitions are basically the same as pass rules in evaluation transitions. All these rules do not create new names or modify existing names, and they all respect the name permutation ψ while creating vectors.

Propagation transitions. Propagation transitions are basically dataflow transitions. Therefore, all these rules do not create new names or modify existing names and they all respect the name permutation ψ while creating vectors.

The other direction (2) can be proved similarly. □

Since $\overset{\alpha}{\sim}$ is a bi-simulation, and thus it is also an equivalence relation. From now on, we will identify machine states modulo name equivalence. The other source of non-determinism comes from propagation mode, where the machine can choose different prop-tokens to proceed. We will prove that these transitions are actually confluent, and thus the machine is deterministic in Section .6.5. But before going into details, we first need to define some general properties of graphs.

6.4 Validity

In this section, we will discuss an important property, namely *validity*, of machine states which represents a collection of invariants which are preserved during execution. These invariants will help us to prove other theorems such as determinism and to eliminate any impossible cases when doing so.

6.4.1 Valid graphs

There are two important properties amount to a *valid graph*: name requirements (Def. 6.4.1) and circularity requirements (Def. 6.4.2).

Definition 6.4.1 (Name requirements)**.** A graph G and a size mapping ζ meets the *name requirements* if it satisfies the followings:

1. for any name a in G, it is either *free* or *bound* and the corresponding F_a-node or P_a-node is unique;
2. size constraint - i) for every $\vec{v}^a$-node in the graph, $\vec{v}^a \in \mathbb{F}^{\zeta(a)}$; ii) for every π_i^a-node, $0 \leq i \leq \zeta(a) - 1$; and iii) for every P_a-node it must have $\zeta(a)$ out-ports and be in the form of:

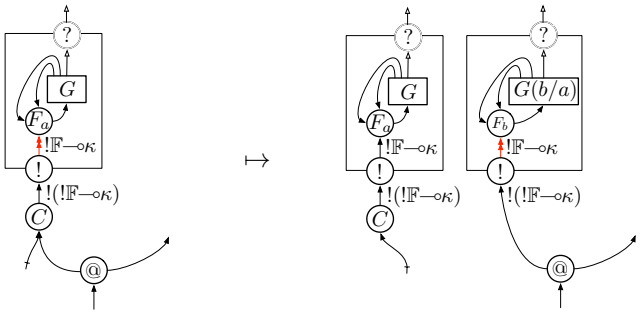
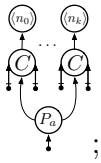


Figure 6.2: δ-rewrite on F_a-box



; and

3. scope - any bound name a must exist only within the !-box that encapsulates the F_a-node, excluding the in-port and the right out-port of the F_a-node, and the in-port of any ?-nodes of the said !-box.

The name requirements ensure that all vectors of the same name family have the same size so that they can be processed correctly by vector operations. The particular shape of the P_a-node is to ensure the P_a-pass rule can be applied to it. Furthermore, it restricts the existence of any bound name to be only within the fusion-scope. This is crucial for the δ-rewrite where a !-box is copied and renamed by some permutation, to make sure that the type of the newly-copied !-node matches the outside. Consider the example shown in Fig. 6.2, since by the name requirement, the return type of the function κ cannot contain the name a. After being copied, the !-box can still match the type of the @-node. We will prove that these requirements are preserved by transitions.

Lemma 6.4.1 (Preservation of name requirements)**.** *For any state π, if π meets the name requirements and $\pi \mapsto \pi'$ then π' also meets the name requirements.*

Proof. This is proved by inspecting all transitions one-by-one.

The δ-rewrite and μ-rewrite create fresh names for the copies, as such they preserve the uniqueness of names. Since they simply copy all the $\vec{v}^a$-nodes and π_i^a-nodes and a box does not contain any P_a-node inside, therefore they also preserve the size constraint. These rules also do not change the shape and the return type of box, therefore they preserve condition (3).

The Fuse-rewrite turns an F_a-node to a P_a-node and extend the ζ function accordingly, which preserves the uniqueness of names. Also, the P_a-node is created in the way as shown in condition (2) therefore it preserves condition (2) as well. Since it does not involve any other F_b-nodes therefore condition (3) is also preserved.

For other transitions, they do not create or remove any F_a/P_a-nodes therefore they all preserve condition (1). Condition (2) is also preserved since vector operations are length-preserving and because these transitions do not create C-nodes on top of other C-nodes and thus the shape of P_a-node is preserved. As for the fold operations, the base vectors created are of the same size as $\zeta(a)$. Condition (3) is also met because all other transitions do not involve bound names. □

The second property, *circularity requirements* concerns cycles in the graph which can only exist in certain conditions.

Definition 6.4.2 (Circularity requirements)**.** A graph G satisfies the *circularity requirements* if all its cycles satisfy one of the following conditions:

1. the path includes the right and left output of a $\lambda/\mu/F_a$-node and all the links and nodes in this path are **only** reachable through the respective $\lambda/\mu/F_a$-node; or
2. the path includes a $\{n\}$-node.

We will now prove that these requirements are preserved by transitions.

Lemma 6.4.2 (Preservation of circularity requirements)**.** *For any state π, if π meets the circularity requirements and $\pi \mapsto \pi'$ then π' also meets the circularity requirements.*

Proof. One way to create circular paths during execution is via the l-rewrite which possibly wires the out-port of a $\{n\}$-node to a graph that reaches the said $\{n\}$-node and thus preserves the requirements. The other way to create circular path is when copying a box, therefore it preserves the requirements since copying boxes preserve the shape of the internal nodes. Other transitions are impossible to create cyclic paths, and thus they must preserve the requirements. □

With these two requirements, we now define formally a *valid graph*.

Definition 6.4.3 (Valid graphs)**.** A graph G and a size mapping ζ is *valid* if they meet the *name requirements* and the *circularity requirements*.

Lemma 6.4.3 (Graph validity preservation)**.** *For any state* (G, ζ, δ, P) *where* (G, ζ) *is valid, if* $(G, \zeta, \delta, P) \mapsto (G', \zeta', \delta', P')$ *then* (G', ζ') *is also valid.*

Proof. It simply follows from Lem. 6.4.1 and Lem. 6.4.2. □

6.4.2 Valid states

In this section, we will define the *validity* of a state and prove that it is preserved by transitions.

Definition 6.4.4 (Evaluated ports)**.** A port e is said to be evaluated if the graph reachable by e can be decomposed into EV such that E is a dataflow graph and V is a value dag.

Definition 6.4.5 (Answer sets)**.** We define the answer sets for any $e : \theta$ as follows:

$$
\begin{aligned}
Ans_{e:\mathbb{F}} &= \{(n, -)\} \textit{ where } n \in \mathbb{F} \\
Ans_{e:Cell} &= \{(n, e)\} \textit{ where } n \in \mathbb{F} \\
Ans_{e:!\mathbb{F}} &= \begin{cases} \{(n, g)\} \textit{ where } n \in \mathbb{F} & \textit{if } e \textit{ is connected to a dataflow graph} \\ \{(n, -)\} \textit{ where } n \in \mathbb{F} & \textit{otherwise} \end{cases} \\
Ans_{e:Unit} &= Ans_{e:!Unit} = \{((), -)\} \\
Ans_{e:\kappa_1 \multimap \kappa_2} &= Ans_{e:!(\kappa_1 \multimap \kappa_2)} = \{(\lambda, -)\} \\
Ans_{e:Vec\ a} &= \{(\vec{v}^a, -)\} \textit{ where } \vec{v}^a \in \mathbb{F}^{\zeta(a)} \\
Ans_{e:Prm\ a} &= \{(\vec{v}^a, e)\} \textit{ where } \vec{v}^a \in \mathbb{F}^{\zeta(a)} \\
Ans_{e:!Vec\ a} &= \begin{cases} \{(\vec{v}^a, g)\} \textit{ where } \vec{v}^a \in \mathbb{F}^{\zeta(a)} & \textit{if } e \textit{ is connected to a dataflow graph} \\ \{(\vec{v}^a, -)\} \textit{ where } \vec{v}^a \in \mathbb{F}^{\zeta(a)} & \textit{otherwise} \end{cases}
\end{aligned}
$$

The answer set represents the set of possible return values for every port node. It is used to define the validity of a token.

Definition 6.4.6 (Valid tokens)**.** A token $(e, d, f, X : S, B)$ is valid in a graph G if it satisfies the following conditions:

1. if $d = \uparrow$ then $X = \star$ else $X \in Ans_e$,
2. if $X = (n, e')$ where $e' \in Port_G$ then e' is connected to a $\{n\}$-node, and
3. if $X = (\vec{v}^a, e')$ where $e' \in Port_G$ then e' is connected to a P_a-node.

The first condition regulates the possible value on top of the computation stack. Every token that is going upwards is basically asking for the value of the connected subgraph and thus the top of the stack must be $\star$. Otherwise, it must carry the proper return value of the said subgraph. The second and the third conditions ensure that if the return value contains a port label, then the port must be connected to either a $\{n\}$-node or a P_a-node that is being used for assignment or linking.

Now we define formally the set of valid states in Def. 6.4.7.

Definition 6.4.7 (Valid states)**.** A state $\pi = (G, \zeta, \delta, P)$ is valid if it satisfies the following:

1. (G, ζ) is a valid graph,
2. for any box that contains $\{n\}$-nodes inside then a) it must be a function box, b) all its ?-node must be connected to a dataflow graph; and c) its internal nodes (without the binder) forms a dataflow graph,
3. if $d = \downarrow$, $e : \kappa$ and $f = \square$ then e must be evaluated,
4. for any $\{n\}$-node that is outside a box, its out-port must be evaluated,
5. any outer C-node must either be connected to a dataflow environment or it is inside a value dag,
6. δ and any tokens in P are valid.

The first condition is self-explanatory. The second condition ensures that the function created by fusion is *safe* when applied to some arguments, i.e. after the beta-rule the $\{n\}$-node originally inside the function box will still be connected to a dataflow graph. The third condition ensures that when the token comes back from a subgraph, it must be evaluated. The fourth condition makes sure that every $\{n\}$-node outside a box must be connected to an evaluated graph, which is important when we prove that propagation terminates. The fifth condition is specific to the call-by-value nature of the language, where only dataflow environment or values (!-boxes) can be shared. The last condition simply states that all the tokens must be valid.

The validity of state is defined in this way to keep it short and simple. However, this in turns makes it impossible to prove that all transitions preserve these properties because they do not contain enough information about the stacks. Alternatively, we prove that every state that is originated from an initial state is valid. But before that we first need to prove a useful lemma saying that if a token comes back from a sub-graph with a return value and empty rewrite flag, then it must have travelled to this current position with upward direction

and having $\star$ on the top of the stack asking the value of the said sub-graph (Lem. 6.4.5). And before that, we first need to prove a similar lemma only for propagation sequences (Lem. 6.4.4).

Lemma 6.4.4 (Factorisation of dataflow sequences)**.** *For any valid initial state* $\pi = Init(G, \zeta, e)$ *all possible sequences from* $\pi \mapsto^n (G', \zeta', (e, \uparrow, sp, \star : S, B), Final^P(\vec{X}))$ *where* $Final^P(\vec{X}) \neq \emptyset$ *can be factorised into*

$$\begin{aligned}\pi &\mapsto^i (G', \zeta', (e, \uparrow, sp, \star : S, B), Init^P) \\ &\mapsto^j (G', \zeta', (e, \uparrow, sp, \star : S, B), Final^P(\vec{X}))\end{aligned}$$

where $n = i + j$.

Proof. This is proved by contradiction. Suppose at least one of the sequence $\pi \mapsto^n (G', \zeta', (e, \uparrow, sp, \star : S, B), Final^P(\vec{X}))$ cannot be factorised. That would also mean that the transition $(G', \zeta', (e, \uparrow, \square, \star : S, B), \emptyset) \mapsto (G', \zeta', (e, \uparrow, sp, \star : S, B), Init^P(\vec{X}))$ will not appear in that sequence. That transition is the only way to trigger propagation from e. Without that the initial state would have non-empty set of prop-tokens which leads to contradiction. □

Lemma 6.4.5 (Factorisation of sequence)**.** *For any valid initial state* $\pi = Init(G, \zeta, e)$*, if* $\pi \mapsto^n \pi'$ *then if* $\pi' = (G', \zeta', (e', \downarrow, \square, X : S, B), \emptyset)$ *then all the possible sequences can be factorised into* $\pi \mapsto^i \pi'' = (G'', \zeta'', (e', \uparrow, \square, \star : S, B), \emptyset) \mapsto^j \pi'$ *such that* $n = i + j$.

Proof. This is proved by induction on the number of transitions (n). For the base case when $n = 0$, (1) is impossible since the direction of the initial state is $\uparrow$.

For evaluation transitions, if the last transition is one of $\{\lambda, (), F_a\}$-pass rules they are straightforward since the last transition itself goes like $(node, \zeta', (i, \uparrow, \square, \star : S, B), \emptyset) \mapsto (node, \zeta', (i, \downarrow, \square, X : S, B), \emptyset)$ where *node* is the corresponding node and X is the appropriate result. As such, $i = k$ and $j = 1$.

If the last transition is the s-rewrite, it is trivial.

If the last transition is one of the $\{p, p_p, m, s\}$-rewrite rules their last transition all have the shape $(\mathcal{G}[\%], \zeta', (i_0, \downarrow, \%, X : S, B), \emptyset) \mapsto (G', \zeta', (i_0, \downarrow, \square, X' : S, B), \emptyset)$ where $\mathcal{G}[]$ is the

surround graph, and thus we will only show the proof for the p-rewrite rule. Since we have the sequence

$$\pi \mapsto^k \pi_1 = (\mathcal{G}[\%], \zeta', (i_0, \downarrow, \%, X : S, B), \emptyset) \mapsto \pi'$$

we can rewind the sequence to

$$\pi \mapsto^{k-1} \pi_2 = (\mathcal{G}[\%], \zeta', (o_0, \downarrow, \square, X : S, B), \emptyset) \mapsto \pi_1 \mapsto \pi'$$

since that is the only possible transition to get to π_1. Now by using the induction hypothesis of (1) on $k-1$ we have that

$$\pi \mapsto^u \pi_3 = (\mathcal{G}'[\%], \zeta', (o_0, \uparrow, \square, \star : S, B), \emptyset) \mapsto^v \pi_2 \mapsto \pi_1 \mapsto \pi'$$

such that $k - 1 = u + v$. In order for the initial state to reach π_3, it must have gone through a state $\pi_4 = (\mathcal{G}''[\%], \zeta', (i_0, \uparrow, \square, \star : S, B), \emptyset)$ otherwise the token would always be on top of the p-node and is never started from an initial state. Therefore, we have that

$$\pi \mapsto^p \pi_4 = (\mathcal{G}''[\%], \zeta', (i_0, \uparrow, \square, \star : S, B), \emptyset) \mapsto^q \pi_3 =\mapsto^v \pi_2 \mapsto \pi_1 \mapsto \pi'$$

such that $u = p + q$. There it gives the factorisation with $i = p$ and $j = q + v + 2$.

If the last transition is one of the $\{op, a, l, a_p\}$-rewrite they are all similar as the token travels through the arguments and comes back to their in-port. We will show the proof for the l-case. Since we have that

$$\pi \mapsto^k \pi_1 = (\mathcal{G}[l], \zeta', (i_0, \downarrow, l(i), Z : S, B), \emptyset) \mapsto \pi'$$

we can rewind the sequence to

$$\pi \mapsto^{k-1} \pi_2 = (\mathcal{G}[l], \zeta', (o_0, \downarrow, \square, X : Y : S, B), \emptyset) \mapsto \pi_1 \mapsto \pi'$$

As such, we can apply the induction hypothesis of (1) on $k-1$ to get

$$\pi \mapsto^u \pi_3 = (\mathcal{G}'[l], \zeta', (o_0, \uparrow, \square, \star : Y : S, B), \emptyset) \mapsto^v \pi_2 \mapsto \pi_1 \mapsto \pi'$$

such that $k - 1 = u + v$. Continue to rewind the sequence, we have that

$$\pi \mapsto^p \pi_4 = (\mathcal{G}''[l], \zeta', (o_1, \downarrow, \square, Y : S, B), \emptyset) \mapsto^q \pi_3 \mapsto^v \pi_2 \mapsto \pi_1 \mapsto \pi'$$

such that $u = p + q$. Again by using the induction hypothesis of (1) on p we have

$$\pi \mapsto^r \pi_5 = (\mathcal{G}'''[l], \zeta', (o_1, \uparrow, \square, \star : S, B), \emptyset) \mapsto^s \pi_4 \mapsto^q \pi_3 \mapsto^v \pi_2 \mapsto \pi_1 \mapsto \pi'$$

such that $p = r + s$. Continue to rewind the sequence, we have that

$$\pi \mapsto^z \pi_6 = (\mathcal{G}''''[l], \zeta', (i_0, \uparrow, \square, \star : S, B), \emptyset) \mapsto^t \pi_5 \mapsto^s \pi_4 \mapsto^q \pi_3 \mapsto^v \pi_2 \mapsto \pi_1 \mapsto \pi'$$

such that $r = z + t$. Therefore, it gives the factorisation with $i = z$ and $j = t + s + q + v + 2$.

For dataflow transitions, if the last transition is one of $n, \vec{v}^a, \{n\}, \langle n \rangle, P_a$-pass rules they are straightforward since the last transition itself goes like $(nd, \zeta', (i, \uparrow, \square, \star : S, B), \emptyset) \mapsto (nd, \zeta', (i, \downarrow, \square, X : S, B), \emptyset)$ where nd is the corresponding nodes and X is the appropriate results. As such, $i = k$ and $j = 1$.

If the last transition is one of $\{d, d_p, !, \pi_i^a\}$-rules, then it is similar to the p-rewrite rule discussed above, and the only difference is that no actual rewrite occurs.

If the last transition is one of $\{\$, \boxplus, \boxtimes, \boxdot\}$-rules, they are all similar to the rewrite rules.

If the last transition is one of the if-rules, since we have that

$$\pi \mapsto^k (\mathcal{G}[if], \zeta', (o_2, \downarrow, \square, X : Y : if_0 : S, B), \emptyset) \mapsto \pi'$$

, we can rewind the sequence to get

$$\begin{aligned}
\pi \mapsto^w \pi_7 &= (\mathcal{G}''''''[if], \zeta''''''', (i_0, \uparrow, \square, \star : S, B), \emptyset) \\
&\mapsto^y \pi_6 = (\mathcal{G}'''''[if], \zeta'''''', (o_0, \uparrow, \square, \star : S, B), \emptyset) \\
&\mapsto^x \pi_5 = (\mathcal{G}''''[if], \zeta''''', (o_0, \downarrow, \square, n : S, B), \emptyset) && \textit{from IH} \\
&\mapsto^t \pi_4 = (\mathcal{G}'''[if], \zeta'''', (o_1, \uparrow, \square, \star : if_0 : S, B), \emptyset) \\
&\mapsto^s \pi_3 = (\mathcal{G}''[if], \zeta''', (o_1, \downarrow, \square, Y : if_0 : S, B), \emptyset) && \textit{from IH} \\
&\mapsto^q \pi_2 = (\mathcal{G}'[if], \zeta'', (o_2, \uparrow, \square, \star : Y : if_0 : S, B), \emptyset) \\
&\mapsto^v \pi_1 = (\mathcal{G}[if], \zeta', (o_2, \downarrow, \square, X : Y : if_0 : S, B), \emptyset) && \textit{from IH} \\
&\mapsto \pi'
\end{aligned}$$

If the last transition is the C-rule then we have that

$$\begin{aligned}
\pi &\mapsto^{p} \pi_3 = (\mathcal{G}'''[C], \zeta''', (i_k, \uparrow, \Box, \star : S, B), \emptyset) \\
&\mapsto^{q} \pi_2 = (\mathcal{G}''[C], \zeta'', (o_0, \uparrow, \Box, \star : S, i_k : B), \emptyset) \\
&\mapsto^{v} \pi_1 = (\mathcal{G}'[C], \zeta', (o_0, \downarrow, \Box, X : S, i_k : B), \emptyset) \qquad \textit{from IH} \\
&\mapsto \pi' = (\mathcal{G}[C], \zeta', (i_k, \downarrow, \Box, X : S, B), \emptyset
\end{aligned}$$

For the propagation transitions, the mode-switching transition (2), where the last transition is $(\mathcal{G}[s], \zeta', (i_0, \uparrow, sp, \star : S, B), \mathit{Final}^P(\vec{X})) \mapsto (\mathcal{G}'[s], \zeta', (i_0, \downarrow, s, (b, -) : S, B), \emptyset)$. If $\mathit{Final}^P(\vec{X}) \neq \emptyset$ then, by using Lem. 6.4.4, we have that

$$\begin{aligned}
\pi &\mapsto^{u} (\mathcal{G}[s], \zeta', (i_0, \uparrow, sp, \star : S, B), \mathit{Init}^P) \\
&\mapsto^{v} (\mathcal{G}[s], \zeta', (i_0, \uparrow, sp, \star : S, B), \mathit{Final}^P(\vec{X})) \\
&\mapsto (\mathcal{G}'[s], \zeta', (i_0, \downarrow, s, (b, -) : S, B), \emptyset)
\end{aligned}$$

where $k = u + v$. Continue to rewind the sequence we have that

$$\begin{aligned}
\pi &\mapsto^{u-1} (\mathcal{G}[s], \zeta', (i_0, \uparrow, \Box, \star : S, B), \emptyset) \\
&\mapsto (\mathcal{G}[s], \zeta', (i_0, \uparrow, sp, \star : S, B), \mathit{Init}^P) \\
&\mapsto^{v} (\mathcal{G}[s], \zeta', (i_0, \uparrow, sp, \star : S, B), \mathit{Final}^P(\vec{X})) \\
&\mapsto (\mathcal{G}'[s], \zeta', (i_0, \downarrow, s, (b, -) : S, B), \emptyset)
\end{aligned}$$

and thus the factorisation with $i = u - 1$ and $j = v + 2$. If $\mathit{Final}^P(\vec{X}) = \emptyset$ then we simply have that $\pi \mapsto^{k-1} (\mathcal{G}[s], \zeta', (i_0, \uparrow, \Box, \star : S, B), \emptyset) \mapsto (\mathcal{G}[s], \zeta', (i_0, \uparrow, sp, \star : S, B), \emptyset) \mapsto (\mathcal{G}'[s], \zeta', (i_0, \downarrow, s, (b, -) : S, B), \emptyset)$ and thus the factorisation with $i = k - 1$ and $j = 2$. □

In below, we prove that validity is preserved by transitions if a state is originated from a valid initial state.

Lemma 6.4.6 (Validity preservation)**.** *For any valid initial state $\pi = \mathit{Init}(G, \zeta, e)$ if $\pi \mapsto^n \pi'$ then π' is also valid.*

Proof. This is proved by induction on the sequence $\pi \mapsto^n \pi'$. For the base case where $n = 0$ we have $\pi' = \pi$ and thus π' is also valid. As for the inductive case where $n = k+1$, i.e. $\pi \mapsto^k \pi^* \mapsto \pi'$, we first apply the induction hypothesis on k we have that second last state is valid, and now we will do a case analysis on all possible transitions. Let $\pi' = (G', e', d', f', S', B', P')$, the first condition follows from Lem. 6.4.3. For condition (2) the only transitions that would create a box is the δ-rewrite, μ-rewrite and the Fuse-rewrite. For the first two, since the box already meets condition (2) and also adding C-node under a dataflow graph would still produce a dataflow graph, therefore these cases are true. As for the Fuse-rewrite since the argument graph was evaluated already and has $\mathbb{F}$ type therefore it is either a dataflow graph or a constant. Therefore, the resulting graph also meets the second condition.

For evaluation transitions For all transitions in this class, P must be valid, since $P = \emptyset$. For all the pass rules, they all satisfy conditions (3) to (4) because the underlying graph does not change at all. As such, we only need to show that the main token δ is valid. If $d' = \uparrow$ these cases are true since we only need to show $S' = \star : S''$ for some S'', all these cases either inherit from π^* or they explicitly put a $\star$ on the stack.

For the $\lambda, (), F_a$-pass rules, they are true as they simply return the value of itself.

For the $p, p_p, m, a, a_p, l, \$, \boxplus, \boxtimes, \boxdot, f$-cases, since $f \neq \square$ we only need to show that the first condition of *valid tokens* is true, i.e. $S' = X : S''$ such that $X \in Ans_{e'}$ which are true.

For the rewrite rules, the Beta-rule indeed satisfies condition (4) because of Lem. 6.4.5, we know that the right-out-port of the @-node must be evaluated, therefore after the rewrite all the $\{n\}$-node must be evaluated. δ must be valid since we have $d = \uparrow$ and $X = \star$.

For the s-rule, p-rule they left behind a constant box which satisfies condition (3). They also meet condition (4) of valid states, as they do not create or change any $\{n\}$-nodes nor evaluated ports.

For the m-rule, r-rule and l-rule, we need to prove that after the rewrites, the $\{n\}$-node is still connecting to a dataflow graph or a constant !-box. We will show the proofs for l-rule.

By using Lem. 6.4.5, we know that the token must have passed through the right out-port of the l-node with $d = \downarrow$ and $f = \square$. As such, its right out-port must be evaluated and since by the induction hypothesis, the evaluated port stays evaluated all the time and thus when the token comes back to the in-port, the right out-port is still evaluated. Since the type of the right out-port is $\mathbb{F}$ there, it could only be connected to a dataflow graph or a constant !-box. Therefore, condition (4) is true after the rewrites.

For the op-rule, a-rule, a_p-rule, if-rule, f-0, f-n-rule, they also meet condition (4) of valid states as they do not create or changes any $\{n\}$-nodes nor evaluated ports.

For the C-c-rule, C-!-rule, δ-rule, μ-rule, the changes are local and does not affect the rest of the graph.

For dataflow transitions For dataflow transitions since the graph is unchanged we only need to show that the main token is valid. Which follows the same idea as the pass rules of evaluation transitions.

For propagation transitions For the two mode switching transitions, (1) is valid since the initial prop-tokens are all valid and (2) is also valid because the rewrite only changes the value inside $\{n\}$-nodes. As for propagation transitions, since all the dataflow rules preserve token validity, therefore they all preserve state validity. □

Lemma 6.4.7 (Validity of initial states)**.** *For any well-typed term* $\Gamma \vdash t : \tau$*,* $\mathit{Init}(\llbracket \Gamma \vdash t : \tau \rrbracket, \emptyset, e)$ *is valid where* e *is the root of the translated graph.*

Proof. This is proved by induction on the type judgement $\Gamma \vdash t : \tau$. □

In below, we prove that if we have a sequence of transitions from $\pi \mapsto^* \pi'$ then we can append any list of elements to the stacks and obtain a similar sequence which does not affect the transitions.

Lemma 6.4.8 (Stack extension over transitions)**.** *For any transitions*

$$(G, \zeta, (e, d, f, S, B), \emptyset) \mapsto (G', \zeta', (e', d', f', S', B'), \emptyset)$$

we also have for any S_e, B_e*,*

$$(G, \zeta, (e, d, f, SS_e, BB_e), \emptyset) \mapsto (G', \zeta', (e', d', f', S'S_e, B'B_e), \emptyset).$$

Proof. This is proved by case analysis on all transitions. □

Corollary 6.4.8.1 (Termination extending over stacks)**.** *For any graph* $G : \Lambda \to \emptyset$ *and any link* $e : \kappa$ *inside* G*, if* $(G, \zeta, (e, d, f, S, B), \emptyset) \mapsto^* (G', \zeta,' (e', d', f', S', B'), \emptyset)$ *then for any* S_e, B_e *we also have*

$$(G, \zeta, (e, d, f, SS_e, BB_e), \emptyset) \mapsto^* (G', \zeta,' (e', d', f', S'S_e, B'B_e), \emptyset).$$

6.5 Determinism

The only source of non-determinism left is that the machine can choose which prop-token to advance during propagation. In this section we prove that propagation always terminates and produces unique results and hence the transitions are deterministic up to equivalence of propagation sequences. But first we need to prove that dataflow graphs always terminate due to their special structure.

Proposition 6.5.1 (Termination of dataflow graphs)**.** *For any valid dataflow graph* $E : \Lambda \to \emptyset$ *and any* $e \in dom(\Lambda)$*, there exists some* $X \in Ans_e$ *such that*

$$Init(E, \zeta, e) \mapsto^* Final(E, \zeta, e, X)$$

and there is no rewrite transition in the sequence.

Proof. This can be proved by induction on the maximum length ($n > 0$) of the path from e that stops at any $\{n\}$-node or $\langle n \rangle$-node.

Base Case ($n = 1$) The only case here would be a single $\{n\}$-node or a $\langle n \rangle$-node and in both cases they terminate.

Inductive Step ($n = k + 1$) This is proved by case analysis on the possible nodes connected by e.

Sub-Cases when e is connected to a \$-node or `if`-node or C-node or d-node or d_p-node or $\boxplus$-node or $\boxtimes$-node or $\odot$-node or π_i^a-node. These cases are similar in the way that the tokens travelled all of their arguments and come back to the in-port, and thus we will show the proofs for \$-node. Running an execution, we have:

$$G\$, \zeta, (e, \uparrow, \square, \star : \square, \square), \emptyset \mapsto G\$, \zeta, (e_1, \uparrow, \square, \star : \square, \square), \emptyset \tag{6.1}$$

$$\mapsto^* G\$, \zeta, (e_1, \downarrow, \square, X : \square, \square), \emptyset \tag{6.2}$$

$$\mapsto G\$, \zeta, (e_2, \uparrow, \square, \star : X : \square, \square), \emptyset \tag{6.3}$$

$$\mapsto^* G\$, \zeta, (e_2, \downarrow, \square, Y : X : \square, \square), \emptyset \tag{6.4}$$

$$\mapsto G\$, \zeta, (e, \downarrow, \$, (m, g) : \square, \square), \emptyset \tag{6.5}$$

$$\mapsto G\$, \zeta, (e, \downarrow, \square, (m, g) : \square, \square), \emptyset \tag{6.6}$$

From (1) to (2) and (3) to (4), since the whole graph is a dataflow graph, therefore at least one of the outputs must be a dataflow graph as well. On the other hand, if the output is connected to a constant !-box then it obviously comes back, therefore we apply the induction hypothesis on the remaining graph and Cor. 6.4.8.1. From (4) to (5) by the definition of dataflow graph, one of the outputs must be connected to a dataflow graph, as such by *validity*, one of Y, X must be in the form of (x, g) and the final result must be (m, g) denoting it is connecting to a dataflow graph.

Sub-Case when e is connected to a P_a-node. This case is trivial as it simply returns the vector of the P_a-node. □

In below, we prove that with only one prop-token, propagation will always terminate because all the $\{n\}$-nodes are connecting to dataflow graphs. As a result, propagation that has multiple tokens will also terminate and produce the same result because sequence of executing dataflow graphs do not contain rewrites.

Lemma 6.5.2 (Termination of prop tokens)**.** *For any valid state*

$$(G, \zeta, (r, \uparrow, sp, S, B), \{e, \uparrow, \star : \square, \square\})$$

where r is the input of an s-node, and $e : \kappa$ of a $\{n\}$-node, there exists a sequence of transitions

$$(G, \zeta, (r, \uparrow, sp, S, B), \{e, \uparrow, \star : \square, \square\}) \mapsto^* (G, \zeta, (r, \uparrow, sp, S, B), \{e, \downarrow, X : \square, \square\})$$

Proof. Since, by *validity*, any $\{n\}$-node will always be connecting to a dataflow graph. Therefore, this is simply a consequence of Prop. 6.5.1. □

Now we can prove that propagation with multiple tokens indeed terminates.

Proposition 6.5.3 (Termination of propagation)**.** *For any valid state*

$$\pi = (G, \zeta, (e, \uparrow, sp, S, B), \mathit{Init}^P)$$

any possible sequences from π will eventually reach $(G, \zeta, (e, \uparrow, sp, S, B), \mathit{Final}^P(\vec{X}))$ for some $\vec{X}$ where e is the input of an s-node.

Proof. This is proved by contradiction. If the sequence does not reach the final state, there are two possibilities: 1) the sequence is infinite or 2) the sequence reaches a state which is not final but cannot proceed. For the first case, it means that at least one of sequences of a prop token is infinite. Since there is no rewrite transitions for prop-tokens, they do not interact with each other, and thus it is the same to say that the execution of the said token is non-terminating, which contradicts Lem. 6.5.2. For the second case, it also means that one of the prop token is stuck, which also contradicts Lem. 6.5.2. □

We also want to prove that any propagation sequences will converge to the same state.

Proposition 6.5.4 (Confluence)**.** *For any valid state $\pi = (G, \zeta, \delta, \mathit{Init}^P)$ if $\pi \mapsto (G, \zeta, \delta, P')$ and $\pi \mapsto (G, \zeta, \delta, P'')$ then they will converge to $(G, \zeta, \delta, \mathit{Final}^P(\vec{X}))$.*

Proof. This is proved by contradiction. Since by Lem. 6.5.3, we have that

$$(G, \zeta, \delta, \mathit{Init}^P) \mapsto (G, \delta, P') \mapsto^* (G, \zeta, \delta, \mathit{Final}^P(\vec{X}))$$

and $(G, \zeta, \delta, Init^P) \mapsto (G, \delta, P'') \mapsto^* (G, \zeta, \delta, Final^P(\vec{X}'))$. Suppose these two states are different, i.e. $Final^P(\vec{X}) \neq Final^P(\vec{X}')$. That means the return value of at least one of the tokens are not the same in the two states. But since there is no rewrite in dataflow graph, tokens must return with the same value for the same graph. Therefore, $Final^P(\vec{X})$ and $Final^P(\vec{X}')$ must be the same. □

Corollary 6.5.4.1. *There is exactly one unique final state for any sequence from* $(G, \zeta, \delta, Init^P)$.

By using this corollary, all the possible propagation sequences

$$(G, \zeta, (e, \uparrow, sp, S, B), Init^P) \mapsto^* (G, \zeta, (e, \uparrow, sp, S, B), Final^P(\vec{X}))$$

form an equivalence class. Therefore, we can treat the non-deterministic propagation sequence as two deterministic transitions which immediately update the content of cells. The replacement is valid because we have proved that propagation sequences are terminating and confluent, therefore the modified machine has the same behaviour as the original one. By doing this, we can now reason about the machine as a deterministic one, which is much more convenient than in the non-deterministic settings.

Definition 6.5.1 (Step-propagation transitions)**.** We replace the transitions in Def. 5.2.8 by the following transitions respectively:

1. $(G, \zeta, (e, \uparrow, \square, \star : S, B), \emptyset) \mapsto (G, \zeta, (e, \uparrow, sp, \star : S, B), \emptyset)$
2. $(G, \zeta, (e, \uparrow, sp, \star : S, B), \emptyset) \mapsto (G'', \zeta, (e, \downarrow, s, (b, -) : S, B), \emptyset)$ where G'' and b are obtained by executing the step-propagation.

Proposition 6.5.5 (Determinism)**.** *The machine is deterministic if we replace Def. 5.2.8 with Def. 6.5.1.*

Proof. This is a consequence of Prop. 6.5.3 and Prop. 6.5.4. □

In the sections below, whenever we talk about transitions, unless otherwise specifies, we always mean the replaced set of transitions. As a result the set of prop-tokens will always be empty, thus we will represent the machine state as only G, ζ, δ from now on.

6.6 Expansion of Contractions

In this section we proved that any graph G behaves the same way as expanding all its outer contractions (C-nodes that are not inside a box) that are connected to !-boxes by using a special kind of simulation, *one-step simulation*.

6.6.1 One-step simulation

Definition 6.6.1 (One-step simulation)**.** A relation R on states is a one-step simulation if it satisfies the following conditions:

1. if $\pi_1\ R\ \pi_2$ and there is a possible transition for π_1 then there exists two states π_1', π_2' such that $\pi_1 \mapsto^+ \pi_1'$, $\pi_2 \mapsto \pi_2'$ and $\pi_1'\ R\ \pi_2'$,
2. if $\pi_1\ R\ \pi_2$ then if there is no possible transition for π_1 then there is no possible transition for π_2 as well,
3. if $\pi_1\ R\ \pi_2$, then π_1 is a final state if and only if π_2 is a final state.

For a binary relation to be a one-step simulation, i.e. $\pi_1 R \pi_2$, one has to show that if π_1 has a possible transition, then one transition of π_2 corresponds to one or more transitions of π_1. It is specifically designed for simulating rewrite transitions. Later in Sec. 6.6 we will show that contraction rewrites forms a one-step simulation.

Lemma 6.6.1. *For any one-step simulation R if $\pi_1\ R\ \pi_2$ and $\pi_2 \mapsto^* \pi_2'$ then there exists a state π_1' such that $\pi_1 \mapsto^* \pi_1'$ and $\pi_1'\ R\ \pi_2'$.*

Proof. This is proved by induction on the sequence $\pi_2 \mapsto^n \pi_2'$.

Base Case $n = 0$ Then $\pi_2 = \pi_2'$ and since $\pi_1\ R\ \pi_2$ then we can pick $\pi_1' = \pi_1$, and thus it is true.

Inductive Case $n = k+1$ We need to show that there exists some state π_1' such that $\pi_1 \mapsto^+ \pi_1'$ and $\pi_1'\ R\ \pi_2'$. Suppose there is no possible transition from π_1 then because $\pi_1\ R\ \pi_2$ thus

there should not be any transition from π_2 as well which contradicts the assumption that $\pi_2 \mapsto^{k+1} \pi_2'$, there must exist a possible transition for π_1. As a result, because $\pi_1\ R\ \pi_2$ there exists some π_1^*, π_2^* such that $\pi_1' \mapsto^+ \pi_1^*$ and $\pi_1^*\ R\ \pi_2^*$. Since $\mapsto$ is deterministic (Prop. 6.5.5) therefore we have $\pi_2 \mapsto \pi_2^* \mapsto^k \pi_2'$ or $\pi_2 \mapsto^2 \pi_2^* \mapsto^{k-1} \pi_2'$. By using the induction hypothesis, there must exist some π_1' such that $\pi_1^* \mapsto^* \pi_1'$ and $\pi_1'\ R\ \pi_2'$. □

In below we prove that if R is a one-step simulation then the transition closure of R is also a one-step simulation.

Lemma 6.6.2. *For any one-step simulation R if $\pi_1\ R^+\ \pi_2$ and $\pi_2 \mapsto^* \pi_2'$ then there exists a state π_1' such that $\pi_1 \mapsto^* \pi_1'$ and $\pi_1'\ R^+\ \pi_2'$.*

Proof. This proved by induction on $n \geq 1$ for $\pi_1\ R^n\ \pi_2$.

Base Case $n = 1$ This is proved in Lem. 6.6.1.

Inductive Case $n = k + 1$ We need to show that if $\pi_1\ R^k \pi_2\ R\ \pi_3$ and $\pi_3 \mapsto^* \pi_3'$ then there exists some π_1' such that $\pi_1 \mapsto^* \pi_1'$ and $\pi_1\ R^+\ \pi_1'$. Since $\pi_2\ R\ \pi_3$ by using Lem. 6.6.1, there exists some π_2' such that $\pi_2 \mapsto^* \pi_2'$ and $\pi_2'\ R^+\ \pi_3'$. Now by using the induction hypothesis on k, there must exist some π_1' such that $\pi_1 \mapsto^* \pi_1'$ and $\pi_1'\ R^+\ \pi_2'$. Thus, we have $\pi_1'\ R^+\ \pi_3'$. □

Lemma 6.6.3. *For any one-step simulation R its transitive closure R^+ is also a one-step simulation.*

Proof. We need to show that R^+ satisfies the three conditions of a one-step simulation.

Case (1) We need to show that if $\pi_1\ R^+\ \pi_e$ and there is a possible transition for π_1 then there exists two states π_1', π_e' such that $\pi_1 \mapsto^+ \pi_1'$, $\pi_e \mapsto \pi_e'$ and $\pi_1'\ R^+\ \pi_e'$.

Since there must exist an $n \geq 1$ such that $\pi_1\ R^n\ \pi_e$, therefore we can do induction on n. When $n = 1$ it is trivially true because $\pi_1\ R\ \pi_e$. When $n = k+1$ then we have $\pi_1\ R^k\ \pi_2\ R\ \pi_e$. Since $\pi_1\ R^k\ \pi_2$ and there is a possible transition for π_1 therefore by using the induction

hypothesis, there exists two states π_1', π_2' such that $\pi_1 \mapsto^+ \pi_1'$, $\pi_2 \mapsto \pi_2'$ and $\pi_1' \ R^+ \ \pi_2'$. Since $\pi_2 \ R \ \pi_e$ and $\pi_2 \mapsto \pi_2'$, therefore these also exists two states π_2'' and π_e' such that $\pi_2 \mapsto^+ \pi_2''$, $\pi_e \mapsto \pi_e'$ and $\pi_2'' \ R \ \pi_e'$. Then by using Lem. 6.6.2, there must exist some state π_1'' such that $\pi_1' \mapsto^* \pi_1''$ and $\pi_1'' \ R \ \pi_2''$. Thus, $\pi_1'' \ R^+ \ \pi_e'$.

Case (2) We need to show that if $\pi_1 \ R^+ \ \pi_e$ and there is no possible transition from π_1 then so is π_e.

Since there must exist an $n \geq 1$ such that $\pi_1 \ R \ \pi_2 \ R \ \ldots \ R \ \pi_{n-1} \ R \ \pi_e$, therefore we can do induction on n. When $n = 1$ we have $\pi_1 \ R \ \pi_e$ and thus π_e has no possible transition. When $n = k + 1$ we have $\pi_1 \ R \ \pi_2 \ R \ \ldots \ R \ \pi_k \ R \ \pi_e$. Since $\pi_1 \ R \ \pi_2$ means that π_2 has no possible transitions and hence by the induction hypothesis this is true.

Case (3) We need to show that if $\pi_1 \ R^+ \ \pi_e$ then π_0 is a final state if and only if π_e is a final state.

Since there must exist an $n \geq 1$ such that $\pi_1 \ R \ \pi_2 \ R \ \ldots \ R \ \pi_{n-1} \ R \ \pi_e$, therefore we can do induction on n. When $n = 1$ it is trivially true because $\pi_1 \ R \ \pi_e$. When $n = k + 1$ we have $\pi_1 \ R \ \pi_2 \ R \ \ldots \ R \ \pi_k \ R \ \pi_e$. a) π_1 is a final state implies π_e is a final state: Since $\pi_1 \ R \ \pi_2$ means that π_2 is also a final state therefore by the induction hypothesis, π_e is also a final state. The other direction is similar. □

Now we prove that if two states are related by a one-step simulation, then either they both terminate or they do not.

Lemma 6.6.4. *For any states π_1, π_2 and a one-step simulation R such that $\pi_1 \ R \ \pi_2$, then $\pi_1 \mapsto^* \pi_1^e \iff \pi_2 \mapsto^* \pi_2^e$ for some final states π_1^e, π_2^e.*

Proof. For the direction $\pi_1 \mapsto^* \pi_1^e \implies \pi_2 \mapsto^* \pi_2^e$, it can be proved by induction on the number of sequence $\pi_1 \mapsto^n \pi_1^e$.

Base Case $n = 0$. Since $\pi_1 = \pi_1^e$ and $\pi_1 \ R \ \pi_2$, therefore π_2 is a final state.

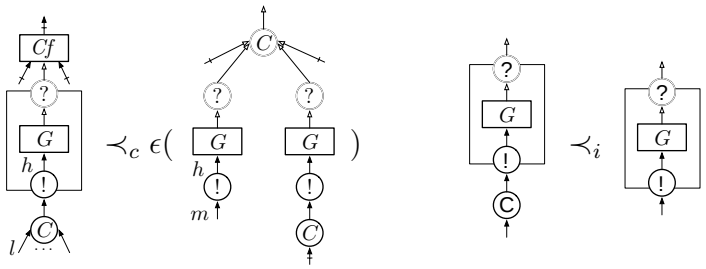
Inductive Step $n = k+1$. We need to show that if $\pi_1 \mapsto^{k+1} \pi_1^e$ then there must exist a final state π_2^e such that $\pi_2 \mapsto^* \pi_2^e$.

We can factorise the sequence into $\pi_1 \mapsto \pi_1' \mapsto^k \pi_1^e$. By the definition of one-step simulation, there must exist two states π_1'' and π_2' such that $\pi_1' \mapsto^* \pi_1''$, $\pi_2 \mapsto \pi_2'$ and $\pi_1''\ R\ \pi_2'$. Therefore, by determinism, either 1) π_1'' is a final state which then by definition π_2' has to be a final state or 2) $\pi_1'' \mapsto^i \pi_1^e$ where $i \leq k$ and thus by induction hypothesis, there must exist some state π_2^e such that $\pi_2' \mapsto^* \pi_2^e$.

For the other direction $\pi_1 \mapsto^* \pi_1^e \impliedby \pi_2 \mapsto^* \pi_2^e$, it is proved in Lem. 6.6.1. □

6.6.2 Expanding contractions

Definition 6.6.2 (Replacement of contractions)**.** We define a relation $\prec$ (replace by) on graphs where $\prec = \prec_c \cup \prec_i$



where Cf is the largest forest of C-nodes connected, the C-node on the left-hand-side of $\prec_c$ has more than one inputs and ϵ is a function that generates fresh names for any free names in G.

Definition 6.6.3. We define a binary relation $\overset{\prec}{\sim}$ on execution states such that for any graph context $\mathcal{G}[\Box]$, $(\mathcal{G}[G_1], \zeta_1, \delta_1) \overset{\prec}{\sim} (\mathcal{G}[G_2], \zeta_2, \delta_2)$ if $G_1 \prec G_2$, or $G_1 = G_2$, $\zeta_1 = \zeta_2$, the position of the main token is in G and $\delta_1 = \delta_2$.

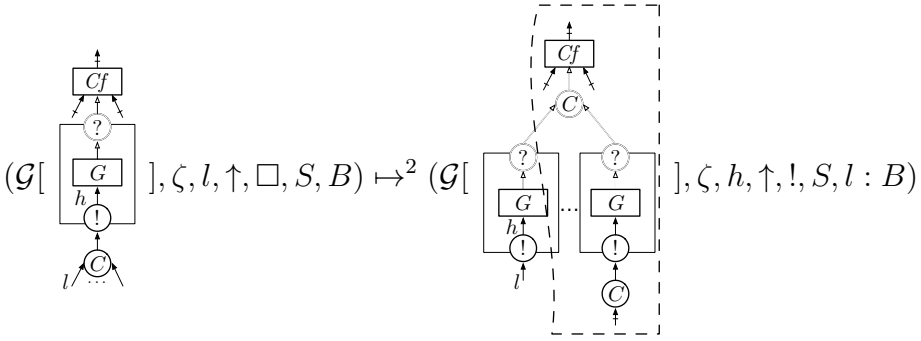
Lemma 6.6.5. $\overset{\prec}{\sim}$ *is a one-step simulation.*

Proof. This is proved by analysing all the possible transitions one-by-one. In some following cases, we will only show the part of the graph that matters.

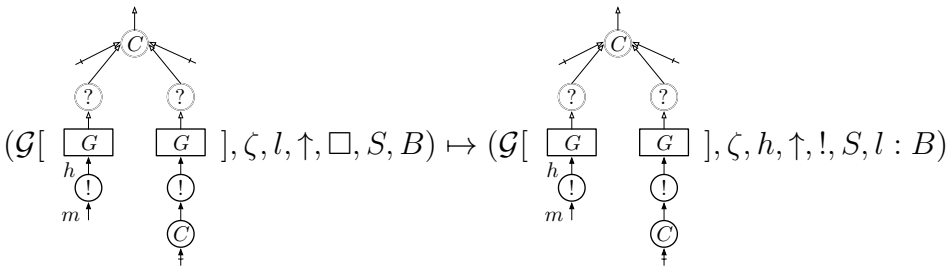
Case when token is inside the context G excluding the border If it is a pass rule or a dataflow transition, then we also have the exact same transition on the right-hand-side. If it is a rewrite transition, then it depends on whether it involves any of the holes of the context. If not, then it is trivial. If yes, then it must be a $C - c$-rewrites involving the forest of C-nodes (Cf) initiated by a !-box on top of it, i.e. $(?H!)[(Cf)(?G_1!)C]$ vs $(?H!)[C(?\vec{G}_2!)]$. In this case, copying the upper !-box using a C-node in Cf will still result in a forest of C-nodes and so it is fine.

Case when the token is at the border of $G_1 \prec_c G_2$. We proved this by analysing all possible cases.

Sub-Case when token is at an input and $d = \uparrow$. We have (1) the token is at one of the inputs of the bottom C-nodes, and (2) the token is at one of the inputs of Cf. For (1) it is straightforward, we have



vs



with G_1' being the outlined subgraph and G_2' be all the boxes except for the left one and the C-node. As for (2) if $f = \square$ then the token simply leaves G_2 while for G_1 the token travels through the forest of C-nodes and then leave. If $f = C$ then there would be rewrites on the

left and then the token leaves G_1.

Sub-Case when the token is at the bottom and $d = \downarrow$. Suppose the token is at one of the inputs of the C-node, by Lem. 6.4.5 there exist a sequence from that input with $d = \uparrow$ trying to evaluate the upper graph. However, if we follow the execution it results in a different graph where the !-box is copied which is not the same as the original graph. But since we have proved that the machine is deterministic, it leads to contradiction. Therefore, the token cannot be at one of the inputs of the C-node, so the only case is when the token is at one of the inputs of Cf. Any possible transitions will take the token back into the ambient graph, and the right-hand-side will have the same transition.

Sub-Case when the token is at the top. By using the same argument in the previous case involving Lem. 6.4.5, the only possible cases would be that the path does not include the !-box nor the C-node. So it would either be a rewrite inside the Cf or a transition that takes the token back to the ambient graph.

Case when token is at the border of $G_1 \prec_e G_2$. The only possible cases are when the token is at the bottom with $d = \uparrow$, and it follows the same idea as previous cases. □

By using one-step simulation, we can now prove that replacing any graph by using $\overset{\prec}{\sim}^+$ preserves the termination property.

Proposition 6.6.6. *For any graphs $H_1, H_2 : \Lambda \to \emptyset$, if $H_1 \prec^+ H_2$ then for any $e \in dom(\Lambda)$, $(H_1, \zeta, e) \Downarrow \iff (H_2, \zeta, e) \Downarrow$ where $\prec^+$ is the transitive closure of $\prec$.*

Proof. Since $Init(H_1, \zeta, e) \overset{\prec}{\sim}^+ Init(H_2, \zeta, e)$ therefore this is simply a consequence of Lem. 6.6.5, Lem. 6.6.3 and Lem. 6.6.4. □

In the following, we show some properties of expanding contractions with value trees and value dags.

Definition 6.6.4 (Value trees)**.** A *value tree* is a tree of !-boxes.

Lemma 6.6.7. *For any value tree* $V : \{e_1 : \kappa\} \to \Lambda$, $VC_2 \prec^+ \vec{C_2}(V \otimes V)$.

Proof. This is proved by induction on the maximum height h of the value tree V (we count any !-box as one big node).

Base Case $h = 0$ This is true as $C_2 = C_2$.

Induction Step $h = k + 1$ We need to show that $(G!)VC_2 \prec^+ \vec{C_2}((G!)V \otimes (G!)V)$ where the maximum height of V is k and $(G!)$ is a value box. By induction hypothesis, we have $(G!)V\vec{C_2} \prec^+ (G!)\vec{C_2}(V \otimes V) \prec_c \vec{C_2}((G!)V \otimes (G!)V)$. □

Lemma 6.6.8. *For any set of value trees* $\vec{V} = V_1 : \{e_1 : \kappa_1\} \to \Lambda_1, \ldots, V_m : \{e_m : \kappa_m\} \to \Lambda_m$, $\vec{V}\vec{C_2} \prec^+ \vec{C_2}(\vec{V} \otimes \vec{V})$.

Proof. This is proved by induction on m using Lem. 6.6.7. □

Lemma 6.6.9. *For any value dag* $V : \{e : \kappa\} \to \Gamma$, *there exists some value tree* V' *such that* $V \prec^+ \vec{C}V'$.

Proof. This is proved by induction on the maximum height of V using similar technique as Lem. 6.6.7. □

6.7 Termination of Recursion-free Programs

In this section, we prove that programs without recursion always terminate. The proof of termination relies on a unary logical relation, also known as a "logical predicate".

Definition 6.7.1 (Logical predicate for terminating graphs)**.** For any term graph $G : \{e : \kappa\} \to \Gamma$, the logical predicate for terminating graphs T_κ is inductively defined on κ as follows:

1. $\kappa \in \{!\mathbb{F}, !\mathit{Unit}, \mathit{Cell}, !\mathit{Vec}\ a, \mathit{Prm}\ a\}$: $G \in T_\kappa$ iff for any dataflow graph $E : \Lambda \to \emptyset$ and ζ such that $\Gamma \subset \Lambda$ and EG, ζ is valid, $(EG, \zeta, e) \Downarrow$.
2. $\kappa = !(\kappa_1 \multimap \kappa_2)$: $G \in T_\kappa$ iff

(a) for any dataflow graph $E : \Lambda \to \emptyset$ and ζ such that $\Gamma \subset \Lambda$ and EG, ζ is valid, $(EG, \zeta, e) \Downarrow$, and

(b) for any term graph $H : \{e_1 : \kappa_1\} \to \Gamma'$ such that $H \in T_{\kappa_1}$, $(G \otimes H)@ \in T_{\kappa_2}$, and

(c) for any dataflow graph $E : \Lambda \to \emptyset$ and ζ such that $\Gamma \subset \Lambda$ and EG, ζ is valid if $\mathit{Init}(EG, \zeta, e) \mapsto^* \mathit{Final}(E'G', \zeta', e, X)$ then for any term graph $H : \{e_1 : \kappa_1\} \to \Gamma'$ such that $H \in T_{\kappa_1}$, $(G' \otimes H)@ \in T_{\kappa_2}$.

We first prove that one-step simulation implies the logical predicate.

Proposition 6.7.1. *For any term graphs $G_1, G_2 : \{e : \kappa\} \to \Gamma$, any one-step simulation R, and for any dataflow graph E and ζ such that EG_1, ζ and EG_2, ζ are valid, $\mathit{Init}(EG_1, \zeta, e)\ R\ \mathit{Init}(EG_2, \zeta, e)$ then $G_1 \in T_\kappa \iff G_2 \in T_\kappa$.*

Proof. The is proved by induction on κ.

Base Case κ is a ground type To show that $G_1 \in T_\kappa \implies G_2 \in T_\kappa$ we need to show that for any dataflow graph E and ζ, $(EG_2, \zeta, e) \Downarrow$. Since $(EG_1, \zeta, e) \Downarrow$ and $\mathit{Init}(EG_1, \zeta, e)\ R\ \mathit{Init}(EG_2, \zeta, e)$, therefore by using Lem. 6.6.6, $(EG_2, \zeta, e) \Downarrow$. The other direction can be shown by similar technique.

Induction Step $\kappa = !(\kappa_1 \multimap \kappa_2)$ To show $G_1 \in T_\kappa \implies G_2 \in T_\kappa$ we need to show that:

(a) for any dataflow graph $E : \Lambda \to \emptyset$ and ζ such that EG_2, ζ is valid $(G_2, \zeta, e) \Downarrow$ which is similar to the base case;

(b) for any term graph $H : \{e_1 : \kappa_1\} \to \Gamma'$ such that $H \in T_{\kappa_1}$, $(G_2 \otimes H)@ \in T_{\kappa_2}$, since $\mathit{Init}(G_1 \otimes H, \zeta, e_2)\ R\ \mathit{Init}(G_2 \otimes H, \zeta, e_2)$, we simply apply the induction hypothesis on κ_2, and;

(c) for any dataflow graph $E : \Lambda \to \emptyset$ such that EG_2, ζ is valid if $\mathit{Init}(EG_2, \zeta, e) \mapsto^* \mathit{Final}(E'G_2', \zeta', e, X)$ then for any term graph $H : \{e_1 : \kappa_1\} \to \Gamma'$ such that $H \in T_{\kappa_1}$,

$(G'_2 \otimes H)@ \in T_{\kappa_2}$. By the definition of one-step simulation, we have that

$$Init(EG_1, \zeta, e) \mapsto^* Final(E'G'_1, \zeta, e, X) and$$

$$Final(E'G'_1, \zeta, e, X)\ R\ Final(E'G'_2, \zeta', e, X)$$

Since $G_1 \in T_\kappa$ therefore $(G'_1 \otimes H)@ \in T_{\kappa_2}$. Therefore, we have $Init(E'(G'_1 \otimes H)@, \zeta, e_2)\ R\ Init(E'(G'_2 \otimes H)@, \zeta, e_2)$ and thus by applying the induction hypothesis, $G'_2 \in T_{\kappa_2}$. The other direction can be shown by similar techniques. □

In below, we prove that adding a set of C-nodes on top of a terminating graph will result in a terminating graph as well.

Lemma 6.7.2. *For any term graph $G : \{e : \kappa\} \to \Gamma$ and any set of parallel C-nodes $\vec{C} : \Gamma \to \omega$,*

$$G \in T_\kappa \implies \vec{C}G \in T_\kappa \tag{6.1}$$

$$\vec{C}G \in T_\kappa \implies G \in T_\kappa \tag{6.2}$$

where $\vec{C}G : \{e : \kappa\} \to \omega$.

Proof. This is proved by induction on κ.

Base Case κ is a ground type To show (1) we need to show that for any dataflow graph E and ζ, $(E\vec{C}G, \zeta, e) \Downarrow$. Since $E\vec{C}$ is still a dataflow graph, by assumption $(E\vec{C}G, \zeta, e) \Downarrow$. The other direction can be shown by similar techniques.

Induction Step $\kappa = !(\kappa_1 \multimap \kappa_2)$ To show (1) we need to show that (a) for any dataflow graph E and ζ, $(E\vec{C}G, \zeta, e) \Downarrow$ which is similar to the base case, (b) for any term graph $H : \{e_1 : \kappa_1\} \to \Gamma'$ such that $H \in T_{\kappa_1}$, $(\vec{C}G \otimes H)@ \in T_{\kappa_2}$ which simply follows from the induction hypothesis and (c) for any dataflow graph $E : \Lambda \to \emptyset$ and ζ such that $E\vec{C}G, \zeta$ is valid if $Init(E\vec{C}G, \zeta, e) \mapsto^* Final(E'G', \zeta, e, X)$ and $\vec{C}G \neq G'$ then for any term graph $H : \{e_1 : \kappa_1\} \to \Gamma'$ such that $H \in T_{\kappa_1}$, $(G' \otimes H)@ \in T_{\kappa_2}$. Since $E\vec{C} \prec^+ F$ for some F and thus we can rewrite the execution as $Init(FG, \zeta, e) \mapsto^* Final(E'G', \zeta, e, X)$. Since $G \in T_\kappa$

therefore by the definition 2(c) of the logical predicate (Def. 6.7.1), $G' \in T_\kappa$. The other direction can be shown by similar techniques. □

In below, we prove that the evaluation of a graph will leave a resulting graph that is stable, meaning that re-evaluation of the graph will give the same result without rewrites.

Definition 6.7.2 (Stable sequences)**.** A sequence is *stable* if it does not contain any rewrite rules.

Lemma 6.7.3. *For any valid graph* $G : (\{e : \kappa\} \cup \Lambda) \to \emptyset, \zeta$ *if* $Init(G, \zeta, e) \mapsto^* (G, \zeta, \delta) \mapsto^R (G', \zeta', \delta')$ *where the first part is stable and* $\mapsto^R$ *denotes a rewrite rule, then we also have a stable sequence* $Init(G', \zeta', e) \mapsto^* (G', \zeta', \delta')$.

Proof. This is proved by case analysis on every rewrite rules $(G, \zeta, \delta) \mapsto^R (G', \zeta', \delta')$.

Cases s-rewrite, p-rewrite, op-rewrite, a-rewrite, l-rewrite, ap-rewrite, C-!-rewrite, δ-rewrite . After the rewrite, it leaves behind a constant box. We will show that case for p-rewrite, we have that $\mathcal{G}[p] = G$ where $\mathcal{G}[]$ is the surrounding graph. Since we have

$$\begin{aligned} Init(\mathcal{G}[p], \zeta, e) &\mapsto^* (\mathcal{G}[p], \zeta, i_0, \downarrow, p, ((), -) : S, B) \\ &\mapsto^R (\mathcal{G}[C \otimes (n!)], \zeta', i_0, \downarrow, \square, ((), -) : S, B) \end{aligned}$$

therefore, by Lem. 6.4.5 we have that

$$\begin{aligned} Init(\mathcal{G}[p], \zeta, e) &\mapsto^* (\mathcal{G}[p], \zeta, i_0, \uparrow, \square, \star : S, B) \\ &\mapsto^* (\mathcal{G}[p], \zeta, i_0, \downarrow, p, ((), -) : S, B) \\ &\mapsto^R (\mathcal{G}[C \otimes (n!)], \zeta', i_0, \downarrow, \square, ((), -) : S, B) \end{aligned}$$

Since $\mathcal{G}[]$ remains unchanged the whole time, therefore we have that

$$\begin{aligned} Init(\mathcal{G}[C \otimes (n!)], \zeta', e) &\mapsto^* (\mathcal{G}[C \otimes (n!)], \zeta', i_0, \uparrow, \square, \star : S, B) \\ &\mapsto^* (\mathcal{G}[C \otimes (n!)], \zeta', i_0, \downarrow, \square, ((), -) : S, B \end{aligned}$$

Cases r-rewrite, r'-rewrite, Beta-rewrite, `if`-n-rewrite, `if`-z-rewrite, f-0-rewrite, f-p-rewrite, Fuse-rewrite, μ-rewrite and Cc-rewrite These cases are similar as these rewrites have the shape $(\mathcal{G}[H], \zeta, i_0, \uparrow, f, \star : S, B) \mapsto^R (\mathcal{G}[H'], \zeta', i_0, \uparrow, \Box, \star : S, B')$. Let us use the `if`-z rewrite as example. Rewinding the sequence we get $\mathit{Init}(\mathcal{G}[H], \zeta, e) \mapsto^* (\mathcal{G}[\texttt{if}], \zeta, o_0, \downarrow, \Box, (0, -) : S, B) \mapsto (\mathcal{G}[\texttt{if}], \zeta, i_0, \uparrow, \texttt{if}, \star : S, B) \mapsto^R (\mathcal{G}[F], \zeta, o_2, \uparrow, \Box, \star : S, B)$. Therefore, by using Lem. 6.4.5 and rewinding the sequence we have that $\mathit{Init}(\mathcal{G}[\texttt{if}], \zeta, e) \mapsto^* (\mathcal{G}[\texttt{if}], \zeta, i_0, \uparrow, \Box, \star : S, B) \mapsto (\mathcal{G}[\texttt{if}], \zeta, o_0, \uparrow, \Box, \star : S, B) \mapsto^* (\mathcal{G}[\texttt{if}], \zeta, o_0, \downarrow, \Box, (0, -) : S, B) \mapsto (\mathcal{G}[\texttt{if}], \zeta, i_0, \uparrow, \texttt{if}, \star : S, B) \mapsto^R (\mathcal{G}[F], \zeta, o_2, \uparrow, \Box, \star : S, B)$. Since $\mathcal{G}[]$ remains unchanged the whole time therefore we also have that $\mathit{Init}(\mathcal{G}[H'], \zeta', e) \mapsto^* (\mathcal{G}[H'], \zeta', i_0, \uparrow, \Box, \star : S, B')$.

Case propagation rewrite The proof is similar. We first have that $\mathit{Init}(\mathcal{G}[s], \zeta, e) \mapsto^* (\mathcal{G}[s], \zeta, e', \uparrow, sp, \star : S, B) \mapsto^P (G', \zeta', e', \downarrow, \Box, (b, -) : S, B)$ then we want to construct a sequence $\mathit{Init}(G', \zeta', e) \mapsto^* (G', \zeta', e', \uparrow, sp, \star : S, B) \mapsto^* (G', \zeta', e', \downarrow, \Box, (b, -) : S, B)$, suppose the first half of the second sequence does not lead to G' which means that there exists some rewrite transitions. However, since the difference between G and G' is that the content of some cells are changed. This would mean that this transitions will also happen in the original sequence, which draws a contradiction. □

Proposition 6.7.4. *For any valid graph $G : \{e : \kappa\} \cup \Lambda \to \emptyset, \zeta$, if $\mathit{Init}(G, \zeta, e) \mapsto^* (G', \zeta', \delta)$ for some G', ζ', δ, then we also have that $\mathit{Init}(G', \zeta', e) \mapsto^* (G', \zeta', \delta)$.*

Proof. Since the sequence $\mathit{Init}(G, \zeta, e) \mapsto^* (G', \zeta', \delta)$ is finite, therefore this can be proved by induction on the number (n) of actual rewrite transitions in this sequence.

Base Case $n = 0$ this means that there are no actual rewrites in the sequence and therefore $G = G'$.

Inductive Step $n = k + 1$ First we factorise the sequence into

$$\mathit{Init}(G, \zeta, e) \mapsto^* (G, \zeta, \delta') \mapsto^R (G'', \zeta'', \delta'') \mapsto^* (G', \zeta', \delta)$$

where $\mapsto^R$ denotes a rewrite rule.

By using Lem. 6.7.3 on $\mathit{Init}(G, \zeta, e) \mapsto^* (G, \zeta, \delta') \mapsto^R (G'', \zeta'', \delta'')$, we have that $\mathit{Init}(G'', \zeta'', e) \mapsto^* (G'', \zeta'', \delta'')$. Since the machine is deterministic we have $\mathit{Init}(G'', \zeta'', e) \mapsto^* (G'', \zeta'', \delta'') \mapsto^* (G', \zeta', \delta)$ with k rewrites transitions. Therefore by using the induction hypothesis on k, we have that $\mathit{Init}(G', \zeta', e) \mapsto^* (G', \zeta', \delta)$. □

Lemma 6.7.5 (Forward reasoning)**.** *For any term graph $G : \{e : \kappa\} \to \Gamma$, any dataflow graph $E : \Lambda \to \emptyset$ and ζ such that $\Gamma \subset \Lambda$, if $\mathit{Init}(EG, \zeta, e) \mapsto^* \mathit{Final}(E'G', \zeta', e, X)$ for some X, ζ' and $G \in T_\kappa$ then $G' \in T_\kappa$.*

Proof. This is proved by induction on κ.

Base case when κ is a ground type Then by *validity* G' must either be empty or a constant !-box and thus $G' \in T_\kappa$.

Inductive case when $\kappa = !(\kappa_1 \multimap \kappa_2)$ We need to show that (a) for any dataflow graph $E : \Lambda \to \emptyset$ and ζ such that FG', ζ is valid, $(FG', \zeta, e) \Downarrow$ which is straightforward as G' must be a !-box, and (b) for any term graph $H : \{e_1 : \kappa_1\} \to \Gamma'$ such that $H \in T_{\kappa_1}$, $(G' \otimes H)@ \in T_{\kappa_2}$ which simply follows from $G \in T_\kappa$. For condition (c), since the graph G' is stable, i.e. any execution from G' will remain unchanged, it follows from the above that for any term graph $H : \{e_1 : \kappa_1\} \to \Gamma'$ such that $H \in T_{\kappa_1}$, $(G' \otimes H)@ \in T_{\kappa_2}$. □

Lemma 6.7.6 (Backward reasoning)**.** *For any term graph $G : \{e : \kappa\} \to \Gamma$, if there exists a graph $G' : \{e : \kappa\} \to \Gamma$ such that (I) for any dataflow graph $E : \Lambda \to \emptyset$ and ζ such that $\Gamma \subset \Lambda$, $\mathit{Init}(EG, \zeta, e) \mapsto^* (E'\vec{V}G', \zeta', \delta)$ for some E', $\vec{V}$ and δ and (II) for any $\vec{V}$ such that $\vec{V}G' \in T_\kappa$ and $V_i \in T_{\kappa_i}$ for some κ_i, then $G \in T_\kappa$.*

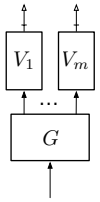
Proof. This is proved by induction on κ.

Base Case κ is a ground type Then we only need to prove that it terminates. Assuming a dataflow environment E and ζ, because of the assumption of G, we would have

$\mathit{Init}(EG, \zeta, e) \mapsto^* (E'\vec{V}G', \zeta', \delta)$. Also by assumption we have that $\vec{V}G' \in T_\kappa$ and thus $(E'\vec{V}G', \zeta', e) \Downarrow$. Since the machine is deterministic (Prop. 6.5.5), and by using Prop. 6.7.4, we have $\mathit{Init}(EG, \zeta, e) \mapsto *(E'\vec{V}G', \zeta', \delta) \mapsto^* \mathit{Final}(E''K, \zeta'', e, X)$ and thus this case is true.

Inductive Case $\kappa = !(\kappa_1 \multimap \kappa_2)$ We need to show that (a) for any dataflow graph $E : \Lambda \to \emptyset$ and ζ such that FG, ζ is valid, $(FG, \zeta, e) \Downarrow$ which is the same as the base case, and (b) for any term graph $H : \{e_1 : \kappa_1\} \to \Gamma'$ such that $H \in T_{\kappa_1}$, $(G \otimes H)@ \in T_{\kappa_2}$. For any dataflow graph F, we would have $\mathit{Init}(F(G \otimes H)@, \zeta, e_2) \mapsto^* (F'(\vec{V} \otimes H'))G', \zeta', \delta)$ for some H' such that $H' \in T_{\kappa_1}$ because of the assumption that G always reduce to G' no matter what the dataflow environment is. This satisfies the assumption (II) of the IH. Therefore, we can apply the IH on κ_2 and get $(G \otimes H)@ \in T_{\kappa_2}$. For (c) if for any dataflow graph $E : \Lambda \to \emptyset$, $\mathit{Init}(EG, \zeta, e) \mapsto^* \mathit{Final}(E''G'', \zeta'', e, X)$ then by determinism we have that $\mathit{Init}(EG, \zeta, e) \mapsto^* (E'\vec{V}G', \zeta', \delta) \mapsto^* \mathit{Final}(E''G'', \zeta'', e, X)$. Then by Prop. 6.7.4, we also have $\mathit{Init}(E'\vec{V}G', \zeta', e) \mapsto^* \mathit{Final}(E''G'', \zeta'', e, X)$. Since $\vec{V}G' \in T_\kappa$, therefore by the definition of the logical predicate $G'' \in T_\kappa$. □

Lemma 6.7.7 (Fundamental lemma)**.** *For any well-typed term $\Gamma \vdash t : \tau$ which is recursion-free, let $G : \{e : \kappa\} \to \Gamma = [\![\Gamma \vdash t : \tau]\!]$, any set of recursion-free value trees $\vec{V} = V_1 : \{e_1 : \kappa_1\} \to \Lambda_1, \ldots, V_m : \{e_m : \kappa_m\} \to \Lambda_m$ such that $V_i \in T_{\kappa_i}$,*



$\in T_\kappa$

where $\vec{V}G : \{e : \kappa\} \to \Lambda$ and $\Lambda = \bigcup \Lambda_i$.

Proof. This is proved by induction on the type derivation of $\Gamma \vdash t : \tau$.

Base Case $\Gamma, x : \tau \vdash x : \tau$ Since G is empty and $V \in T_\kappa$ this case is straightforward.

Base Case $\Gamma \vdash n : \mathbb{F}$ and $\Gamma \vdash \mathtt{pc}\ n : \mathbb{F}$ In these cases since $\vec{V}$ is empty we only need to show that for any dataflow graph $E : \Lambda \to \emptyset$ and ζ such that $\Gamma \subset \Lambda$, $(EG, \zeta, e) \Downarrow$ which are straight-forward.

Inductive Step $\Gamma \vdash \lambda x.t : \tau_1 \to \tau_2$. We need to show that (a) for any dataflow graph $E : \Lambda \to \emptyset$ and ζ such that $\Gamma \subset \Lambda$, $(E\vec{V}G, \zeta, e) \Downarrow$, (b) for any term graph $H : \{e_1 : \kappa_1\} \to \Gamma'$ such that $H \in T_{\kappa_1}$, $(\vec{V}G \otimes H)@ \in T_{\kappa_2}$. Condition (c) simply follows from (b) as the graph $\vec{V}G$ remains unchanged for any execution.

The proof of (a) is straight-forward. In order to show (b), we assume a dataflow graph $E : \Lambda \to \emptyset$ and ζ such that $\Gamma \cup \Gamma' \subset \Lambda$ and run a sequence from $Init(E(\vec{V}G \otimes H)@, \zeta, e_2)$:

$$E(\vec{V}G \otimes H)@, \zeta, (e_2, \uparrow, \Box, \star : \Box, \Box) \tag{6.1}$$

$$\mapsto E(\vec{V}G \otimes H)@, \zeta, (e_1, \uparrow, \Box, \star : \Box, \Box) \tag{6.2}$$

$$\mapsto^* E'(\vec{V}G \otimes H')@, \zeta', (e_1, \downarrow, \Box, X : \Box, \Box) \tag{6.3}$$

$$\mapsto E'(\vec{V}G \otimes H')@, \zeta', (e, \uparrow, \Box, \star : \Box, \Box) \tag{6.4}$$

$$\mapsto E'(\vec{V}G \otimes H')@, \zeta', (a, \uparrow, !, \star : \Box, \Box) \tag{6.5}$$

$$\mapsto E'(\vec{V}G \otimes H')@, \zeta', (a, \uparrow, \Box, \star : \Box, \Box) \tag{6.6}$$

$$\mapsto E'(\vec{V}G \otimes H')@, \zeta', (a, \downarrow, \Box, (\lambda, -) : \Box, \Box) \tag{6.7}$$

$$\mapsto E'(\vec{V}G \otimes H')@, \zeta', (e, \downarrow, \Box, (\lambda, -) : \Box, \Box) \tag{6.8}$$

$$\mapsto E'(\vec{V}(?([\![t]\!]\lambda)!) \otimes H')@, \zeta', (e_2, \uparrow, @, \star : \Box, \Box) \tag{6.9}$$

$$\mapsto E'(\vec{V} \otimes H')[\![t]\!], \zeta', (e_2, \uparrow, \Box, \star : \Box, \Box) \tag{6.10}$$

where $G = ?([\![t]\!])\lambda!$ and a is the in-port of the λ-node. Going from (2) to (3) we apply the condition on $H \in T_{\kappa_1}$ and Cor. 6.4.8.1 and Lem. 6.7.5 and thus H' is a *value dag* and $H' \in T_{\kappa_1}$. Now by using Lem. 6.6.9, we have $H' \prec^+ \vec{C}H^*$ such that H^* a value tree and $\vec{C}H^* \in T_{\kappa_1}$. By using Lem. 6.7.2, $H^* \in T_{\kappa_1}$. Now we can use the induction hypothesis on t to get $(\vec{V} \otimes H^*)[\![t]\!] \in T_{\kappa_2}$. Then by Lem. 6.7.2, we have that $(\vec{V} \otimes \vec{C}H^*)[\![t]\!] \in T_{\kappa_2}$. Thus, by using Prop. 6.7.1, we have $(\vec{V} \otimes H')[\![t]\!] \in T_{\kappa_2}$. Therefore, by using Lem. 6.7.6, we have that

$(\vec{V}G \otimes H)@ \in T_{\kappa_2}$.

Inductive Step $\Gamma \vdash t\ u : \tau$. In this case, we need to show that $\vec{V}G \in T_\kappa$ where $G = \vec{C_2}([\![t]\!] \otimes [\![u]\!])@$.

First by Lem. 6.6.8 we have that $\vec{V}\vec{C_2}([\![t]\!] \otimes [\![u]\!])@ \prec^+ \vec{C_2}(\vec{V}[\![t]\!] \otimes \vec{V}[\![u]\!])@$. Therefore by Prop 6.7.1, it suffices to show that $\vec{C_2}(\vec{V}[\![t]\!] \otimes \vec{V}[\![u]\!])@ \in T_\kappa$. Since by induction hypothesis on t and u, we have that $(\vec{V}[\![t]\!] \otimes \vec{V}[\![u]\!])@ \in T_\kappa$ and then by Lem. 6.7.2, we have that $\vec{C_2}(\vec{V}[\![t]\!] \otimes \vec{V}[\![u]\!])@ \in T_\kappa$.

Base Case $\Gamma \vdash op : \tau$. We will separate all the operations into four groups: nullary and unary operations $\{\texttt{ref}, \texttt{deref}, \texttt{deref}_p, \texttt{step}, \texttt{peek}, \texttt{peek}_p\}$, binary operations with $d = \downarrow$ before rewrite $\{\$, \boxtimes, \boxplus, \texttt{link}, \texttt{assign}, \texttt{assign}_p\}$ and binary operations with $d = \uparrow$ before rewrite $\{\texttt{root}, \texttt{foldp}_p\}$.

Sub-Case $\Gamma \vdash \texttt{ref} : \mathbb{F} \to \mathit{Cell}$. Since `ref` is a closed function, its translated graph G there is no outgoing edges. Therefore, $\vec{V}$ is an empty graph. In this case we need to show that (a) for any dataflow graph $E : \Lambda \to \emptyset$ and ζ such that $\Gamma \subset \Lambda$, $(EG, \zeta, e) \Downarrow$, (b) for any term graph $H : \{e_1 : \mathbb{F}\} \to \Gamma'$ such that $H \in T_{\mathbb{F}}$, $(G \otimes H)@ \in T_{\mathit{Cell}}$. Condition (c) is not needed as the graph remains unchanged during any execution. (a) is trivial, as the graph G is simply a function box. For (b) we assume a dataflow graph E and ζ and run an execution from

$Init(E(G \otimes H)@, \zeta, e_2)$ where e_2 is the in-port of the @-node.

$$Init(E(G \otimes H)@, \zeta, e_2) \mapsto (E(G \otimes H)@, \zeta, e_1, \uparrow, \Box, \star : \Box, \Box) \tag{6.1}$$

$$\mapsto^* (E'(G \otimes H')@, \zeta', e_1, \downarrow, \Box, (n, x) : \Box, \Box) \tag{6.2}$$

$$\mapsto (E'(G \otimes H')@, \zeta', e, \uparrow, \Box, \star : \Box, \Box) \tag{6.3}$$

$$\mapsto^3 (E'(G \otimes H')@, \zeta', e, \downarrow, \Box, (\lambda, -) : \Box, \Box) \tag{6.4}$$

$$\mapsto (E'(G \otimes H')@, \zeta', e_2, \uparrow, @, \star : \Box, \Box) \tag{6.5}$$

$$\mapsto (E'H'm, \zeta', e, \uparrow, \Box, \star : \Box, \Box) \tag{6.6}$$

$$\mapsto (E'H'm, \zeta', e_1, \uparrow, \Box, \star : \Box, \Box) \tag{6.7}$$

$$\mapsto^* (E'H'm, \zeta'', e_1, \downarrow, \Box, (n, x) : \Box, \Box) \tag{6.8}$$

$$\mapsto (E'H'm, \zeta'', e, \downarrow, m, (n, x) : \Box, \Box) \tag{6.9}$$

$$\mapsto (E'H'\{n\}, \zeta'', e, \downarrow, \Box, (n, e) : \Box, \Box) \tag{6.10}$$

From (1) to (2) we use the condition of H and from (7) to (8) it follows from the validity as e_1 is evaluated, and it has to be connected to either a !-box or a dataflow graph.

Sub-Case $\Gamma \vdash \$: \mathbb{F} \to \mathbb{F} \to \mathbb{F}$. In this case, since $\vec{V}$ is empty, we need to show that $G \in T_{!(!\mathbb{F} \multimap !\mathbb{F} \multimap !\mathbb{F})}$. This graph indeed terminates and result in the same graph, therefore we only need to prove that for any $H_1 : \{e_1 : !\mathbb{F}\} \to \Gamma'$, such that $H_1 \in T_{!\mathbb{F}}$, $(G \otimes H_1)@ \in T_{!(\mathbb{F} \multimap \mathbb{F})}$.

To prove the first condition, assuming a dataflow graph E and ζ, we would have a sequence $Init(E(G \otimes H_1)@, e_2) \mapsto^* Final(E'H_1'G, e_2, (\lambda, -))$ by using the condition of H_1 such that $H_1' \in T_{!\mathbb{F}}$ where e_2 is the input of the bottom @-node. Therefore, it terminates, and we then need to prove that $H_1'G \in T_{!(\mathbb{F} \multimap \mathbb{F})}$. This graph indeed terminates and result in the same graph. Therefore, we only need to prove that for any $H_2 : \{e_3 : !\mathbb{F}\} \to \Gamma''$ such that $H_2 \in T_{!\mathbb{F}}$, $(H_1'G \otimes H_2)@ \in T_{!\mathbb{F}}$. Since it is of ground type, therefore we only need to prove that it terminates. Assuming a dataflow graph E and ζ we would have a sequence from $Init(E(H_1'G \otimes H_2)@, \zeta, e_3) \mapsto^* (E'(H_1' \otimes H_2')\$, \zeta', e_3, \downarrow, \$, X : S, \Box)$. Depending on the value

of X, it would transit to either one of the following:

$$E'(H_1' \otimes H_2')\$, \zeta', (e, \downarrow, \Box, n, g : \Box, \Box) \tag{6.11}$$

$$E'(H_1'C \otimes H_2'C)(n)!, \zeta', (e, \downarrow, \Box, (n, -) : \Box, \Box) \tag{6.12}$$

Therefore we have proved that for any dataflow graph, the graph indeed terminates. Since in the first case, e_3 will be connected to a dataflow graph, thus we take the resulting term graph to be the empty graph and thus for any dataflow graph it terminates. As for the second case, a constant always satisfies the logical predicate.

To prove the second condition $(G \otimes H_1)@ \in T_{!(\mathbb{F} \multimap \mathbb{F})}$, we need to prove that for any $H_2 : \{e_3 : !\mathbb{F}\} \to \Gamma''$ such that $H_2 \in T_{!\mathbb{F}}$, $((G \otimes H_1)@ \otimes H_2)@ \in T_{!\mathbb{F}}$. The proof is similar to the case above.

Sub-Cases $\Gamma \vdash \texttt{link} : \mathit{Cell} \to \mathbb{F} \to \mathit{Unit}$, $\Gamma \vdash \texttt{assign} : \mathit{Cell} \to \mathbb{F} \to \mathit{Unit}$ and $\Gamma \vdash \texttt{root} : \mathit{Cell} \to \mathbb{F} \to \mathit{Unit}$. These cases are similar to the \$-case but also needs *validity* to show that the cell being updated exists.

Sub-Cases $(\Gamma \vdash \boxplus : \mathit{Vec}\ a \to \mathit{Vec}\ a \to \mathit{Vec}\ a)$ and $\Gamma \vdash \texttt{assign}_p : \mathit{Prm}\ a \to \mathit{Vec}\ a \to \mathit{Unit}$. These cases are similar to the \$-case but also needs *validity* to show that the vectors are in the same family and thus having the same size.

Sub-Cases $(\Gamma \vdash \boxtimes : \mathbb{F} \to \mathit{Vec}\ a \to \mathit{Vec}\ a)$ and $(\Gamma \vdash \texttt{foldp}_p : (\mathit{Vec}\ a \to \mathit{Vec}\ a \to \mathit{Vec}\ a) \to \mathit{Vec}\ a \to \mathit{Vec}\ a)$. These cases are similar to the above cases.

Inductive Step $\Gamma \vdash \texttt{if}\ t\ \texttt{then}\ t_1\ \texttt{else}\ t_2 : \gamma$ Since only ground types are allowed in the branches, this case follows the same technique shown in the previous case.

Inductive Step $\Gamma \vdash \mathcal{F}(f, x).t : \mathbb{F} \to \tau$ This case follows the same idea as the λ-case. The only difference is that after the Fuse-rewrite, the bound variable wire of $[\![t]\!]$ is now connected to a newly created P_a-node which is indeed a valid dataflow graph and the rest follows exactly the same procedure as the λ-case. □

Proposition 6.7.8 (Termination)**.** *For any closed well-typed term* $- \mid - \vdash t : \tau$ *which is recursion-free,* $\mathit{Init}([\![- \vdash t : \tau]\!], \emptyset, r) \mapsto^* \mathit{Final}(G, \zeta, r, X)$ *for some* G, ζ, X *where* $X \in \mathit{Ans}_r$ *and* r *is the root of* $[\![- \vdash t : \tau]\!]$*.*

Proof. This is just a consequence of the fundamental lemma (Lem. 6.7.7). □

6.8 Type Soundness

In this section, we prove that the language simTSD is type safe, meaning that (1) well-typed recursion-free programs always terminate; and (2) well-typed programs will not get stuck. We first prove that all transitions preserve the interface of the graph.

Proposition 6.8.1 (Interface preservation)**.** *For any state* (G, ζ, δ) *where* $G : \Lambda \to \emptyset$*, all transitions send* G *to another graph* G' *with the same interface, i.e.*

$$(G : \Lambda \to \emptyset) \wedge (G, \zeta, \delta) \mapsto (G', \zeta', \delta') \implies G' : \Lambda \to \emptyset.$$

Proof. All the pass transitions indeed satisfy the statement, as they do not change the shape of the graph at all. As for rewrite transitions, the proof is done by inspecting them one-by-one. □

Corollary 6.8.1.1. *For any state* (G, ζ, δ) *where* $G : \Lambda \to \emptyset$*, if* $(G, \zeta, \delta) \mapsto^* (G', \zeta', \delta')$ *then* $G' : \Lambda \to \emptyset$*.*

Proposition 6.8.2 (Productivity)**.** *For any valid graph* $(G : \{e : \kappa\} \to \emptyset), \zeta$*, if* $\mathit{Init}(G, \zeta, e) \mapsto^n \pi$ *then either* π *is final or there exists a* π' *such that* $\pi \mapsto \pi'$*.*

Proof. Let $\pi = (G', \zeta', (e', d, f, S, B))$, this is proved by inspecting all possible nodes connected by e' one-by-one.

For $d = \uparrow$ **and** $f = \square$ By validity $S = \star : S'$ and there must exist a possible transition for any node that is connected by e'.

For $d = \downarrow$ and $f = \square$ It depends on the node that is connected to e'.

Sub-case if e' is not connected by any nodes, i.e. $e' = e$, then it must be the final state and this is proved by induction on n. When $n = 0$ this is impossible because $d = \downarrow$. When $n = k + 1$ we can prove this by contradiction. Suppose it is not a final state, i.e. either $S = X : S'$ such that $S' \neq \square$ or $B \neq \square$ then by Lem. 6.4.5 we have that $(G'', \zeta'', e', \uparrow, \square, S, B) \mapsto^* (G', \zeta', e', \downarrow, \square, S, B)$. Since the machine is deterministic we have $Init(G, \zeta, e) \mapsto^* (G'', \zeta'', e', \uparrow, \square, S, B)$. By using the induction hypothesis, $S = X : \square$ and $B = \square$ and it leads to contradiction.

Sub-case if it is one of $\lambda, \mu, F_a, \{n\}, P_a$-nodes these cases are not possible because there is no transition that takes a token to their arms, which contradicts Lem. 6.4.5.

Sub-case if it is a ?-node this is impossible because there is no transition that takes a token to a ?-node, which contradicts Lem. 6.4.5.

Sub-case if it is one of $p, d, m, p_p, d_p, \pi_i^a, !, C$-node then the next transition would simply be the corresponding transition that adds an appropriate rewrite flag. For C-node, by Lem. 6.4.5, the token must have come from one of its in-ports and thus a next transition is possible.

Sub-case if e' is the right-outport of $a, l, \boxplus, \boxtimes, \boxdot, a_p, f_p$, `if` then the next transition would simply be going to the left out-port with $\star$ added to the stack.

Sub-case if e' is the left-outport of $a, l, \boxplus, \boxtimes, \boxdot, a_p, f_p$ or the middle-out-port of `if`-node then by using Lem. 6.4.5, the token must have travelled its right out-port already and thus the stack is in the form $X : Y : S'$ or $X : Y : \texttt{if}_{0/1} : S'$. Therefore, the next transition would be taking the token to the in-port of the respective nodes.

Sub-case if e' is the left-outport of a `if`-node then there must be a next transition depending on the stack element.

For $f \neq \square$ Since all the rewrite transitions (except for Cc-rule) only look at the first element of the computation stack, and by validity when $d = \downarrow$ there is always one element in the stack with $X \in Ans_{e'}$ therefore the next transition would simply be the respective rewrite

transition. We also use Lem. 6.4.5 to exclude any cases where the rewrite flag and the node do not match because in order to generate the flag the machine must go through the specific transition. For example let us assume that $\pi = (\mathcal{G}[\neg], \zeta', (i_0, \downarrow, f, X : S, B))$ where i_0 is the input of the $\neg$-node and $f \neq \neg$. Using Lem. 6.4.5, we know that there exist a state such that $\mathit{Init}(G, \zeta, e) \mapsto^* (\mathcal{G}'[\neg], \zeta'', (i_0, \uparrow, \square, \star : S, B)) \mapsto^* (\mathcal{G}[\neg], \zeta', (i_0, \downarrow, f, X : S, B))$. However, if we continue from the middle state, we will get a sequence to $(\mathcal{G}[\neg], \zeta', (i_0, \downarrow, \neg, X : S, B))$ which contradicts our assumption. Therefore, f must be $\neg$. As for the Cc-rule, we also use Lem. 6.4.5 to prove that the box stack must have at least two elements in it because the token must have travelled through these C-nodes before reaching the !-node. □

Proposition 6.8.3 (Progress)**.** *For any closed well-typed term $- \vdash t : \tau$, if $\mathit{Init}([\![- \vdash t : \tau]\!], \emptyset, r) \mapsto^n (G, \zeta, \delta)$ for some G, ζ, δ where r is the root of $[\![- \vdash t : \tau]\!]$, then either (G, ζ, δ) is a final state or there exists another state (G', ζ', δ') such that $(G, \zeta, \delta) \mapsto (G', \zeta', \delta')$.*

Proof. Since by Lem. 6.4.7, $\mathit{Init}([\![- \vdash t : \tau]\!], r)$ is valid, this is simply a consequence of Prop. 6.8.2. □

Theorem 6.8.4 (Type soundness)**.** *For any closed well-typed term $- \vdash t : \tau$, let $G : \{r : [\![\tau]\!]\} \to \emptyset = [\![- \vdash t : \tau]\!]$ then*

1. *if t is recursion-free then the execution from $\mathit{Init}(G, \emptyset, r)$ will terminate,*
2. *if t is with recursion then the execution from $\mathit{Init}(G, \emptyset, r)$ will not get stuck.*

Proof. This is just a consequence of Cor. 6.8.1.1, Prop. 6.7.8 and Prop. 6.8.3. □

6.9 Symmetry

In this section we prove an important property, *symmetry*, which states that given a permutation of vector indices acting on a graph, the resulting graph and token value must respect the permutation as well. It is easy to see that most of the operations such as vector addition are symmetric, however it is not the case with `foldp` because of effects. Consider the term

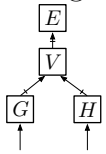
`foldp` ($\lambda b.$`assignp` $ps\ b; b$) ps, since the semantics of `foldp` is to create a graph with a chain of $\boxplus$-node aggregating the result of the function applying to the set of base vectors, but the order of created base vectors are independent of the permutation, and thus it will result in a graph that does not respect the permutation. Therefore, symmetry can only be achieved when the function passed to `foldp` and the context used by it is pure. Converting this requirement in terms of graph, it is to say the reachable graph of an f-node right before the rewrite happens has to be *pure*.

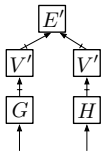
Definition 6.9.1 (Indices)**.** For any vector space $\mathbb{F}^n$, the set of its indices $idx(\mathbb{F}^n)$ is defined as $\{0, \ldots, n-1\}$.

Definition 6.9.2 (Permutation on states)**.** For any machine state $\psi = (G, \zeta, \delta)$, any free name $a \in \mathit{FreeName}_G$ and any permutation on the family $\phi_a : idx(\mathbb{F}^{\zeta(a)}) \to idx(\mathbb{F}^{\zeta(a)})$, $\phi_a \cdot \psi$ replaces every $\vec{v}^a$ in G, δ with $\phi_a \cdot \vec{v}^a$, every π_i^a with $\pi^a_{\phi_a(i)}$ and changes every out-port o_i of P_a-node with $o_{\phi_a(i)}$. Let ϕ be the set of permutations on all the names in ψ, we define $\phi \cdot \psi = \phi_a \cdot \phi_b \cdot \ldots \phi_x \cdot \psi$.

Definition 6.9.3 (Graph equivalence)**.** For any graph G_1, G_2 we say that $G_1 \equiv G_2$ if $\mathit{Init}(G_1, \emptyset, e) \Downarrow \mathit{Final}(G_1', e, \zeta_1, X)$ and $\mathit{Init}(G_2, \emptyset, e) \Downarrow \mathit{Final}(G_2', e, \zeta_2, Y)$ such that $X = Y$.

Lemma 6.9.1 (Commutativity of $\boxplus$)**.** *If the reachable graph from* $\boxplus$ *is pure, then* $\boxplus$ *is commutative, i.e. for any context graph* $\mathcal{G}$, $\mathcal{G}[$  $] \equiv \mathcal{G}[$  $]$.

Proof. Suppose the tokens never travel to the $\boxplus$-node then since the contexts are the same, it must come back with the same token value. Now assume that the tokens did travel to the in-port of the $\boxplus$-node then they must carry the same token data as they have travelled the exact same context graph. At this point the reachable graph U from the $\boxplus$-node can be decomposed into:  where E is a dataflow environment and V is a value dag. This is because if G and H are not isolated, then there must be some part of them that are shared

by the two outputs of the $\boxplus$-node. The shared part must be shared through a C-node. By Def. 6.4.7, an outer C-node must either be connected to a dataflow environment or inside the value dag meaning that the shared part can be considered inside E or V. By Def. 6.6.2, it can be replaced by $U' =$  and since by Lem. 6.6.5 we have that $U\boxplus \prec^+ U'\boxplus$. Since U and U' are pure, therefore any execution on them would not change anything in E'. Therefore, the tokens must come back to the o_0-port of the $\boxplus$-node with the computation stack $X : Y : S$ and $Y : X : S$. After that they transit to the in-port with $\boxplus XY$ and $\boxplus YX$ respectively which are equal. And thus the token leaves the $\boxplus$-node and back to the context with the same token value. $\square$

Definition 6.9.4. For any permutation on states ϕ, we define a binary relation on states $\psi_1 \overset{\phi}{\sim} \psi_2$ if there exists some state ψ_3 such that $G_1 \equiv G_3$ and $\psi_3 = \phi \cdot \psi_2$ where G_1, G_3 are the graphs in ψ_1, ψ_3 respectively.

Lemma 6.9.2. *$\overset{\phi}{\sim}$ is a bi-simulation if for any f-rewrite the reachable graph from the f-node is pure.*

Proof. This is proved by inspecting all the transitions one-by-one and indeed all the transitions do respect the permutation ϕ except for the f-rewrite which creates base vectors in the same order independent to the permutation. However, because of Lem. 6.9.1 there exists a graph G_3 which rewires the out-ports of the $\boxplus$-nodes such that they conform to the permutation and $G_1 \equiv G_3$. $\square$

Theorem 6.9.3 (Symmetry)**.** *For any closed graph $G : \{e : \kappa\} \to \emptyset$ and any permutation on states ϕ, $\mathit{Init}(G, \emptyset, e) \Downarrow \mathit{Final}(G_1, e, \zeta_1, X)$ and $\phi \cdot \mathit{Init}(G, \emptyset, e) \Downarrow \mathit{Final}(G_2, e, \zeta_2, Y)$ and for any f-rewrite in the execution the reachable graph from the f-node is pure then $X = \phi \cdot Y$.*

Proof. This is simply a consequence of Lem. 6.9.2. $\square$

6.10 Graph equivalences

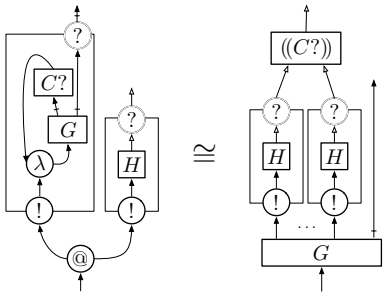
In this section, we will discuss program equivalence and optimisation in the form of *graphs* as opposed to *terms*. First, we define the notion of contextual equivalence as in [62].

Definition 6.10.1 (Graph contexts)**.** A graph context $\mathcal{G}$ is an incomplete graph with a hole inside that can be plugged in with another graph H to form a complete graph $\mathcal{G}[H]$.

Definition 6.10.2 (Graphical contextual equivalence)**.** For any graph G_1, G_2 we say that $G_1 \cong G_2$ if for any context graph $\mathcal{G}$, $\mathcal{G}[G_1] \equiv \mathcal{G}[G_2]$.

The graphical contextual equivalence is indeed an equivalence relation and also a congruence with respect to the graph context.

In the coming sections we will prove beta-equivalence in terms of graph as shown in the following graph equation:



if H is not a μ-box where $C?$ is a tree of  and  and $((C?))$ is a tree of and  such that they have the same structure. The $C?$-graph represents all the instances of the variable used in the function, therefore substitution can be seen as giving a copy of the argument to every leaf in the graph $C?$.

Graphical beta-equivalence can be separated into three parts: (1) *rewiring* the connection between the bound variable and the argument (Sec. 6.10.2), (2) *copying* the box values representing individual substitution for each instance (Sec. 6.10.3) and (3) *engulfing* a box into another box representing substitution through other functions (Sec. 6.10.4). Together these three equivalences imply beta-equivalence, but they are interesting in their own right.

6.10.1 U-simulation

One way to prove contextual equivalence is to make use of the 'locality' of transitions, i.e. transitions only affect a sub-graph, to prove that the execution of one graph can be simulated by another. In particular, we define a special form of simulation: U-simulation.

Definition 6.10.3 (U-simulation)**.** A relation R on states is a U-simulation if it satisfies the following conditions:

1. if $\pi_1\ R\ \pi_2$ and there is a possible transition for π_1, i.e. $\pi_1 \mapsto \pi_1'$ then either
   (a) there exists an $n > 0$ and two states π_1'', π_2' such that $\pi_1 \mapsto^n \pi_1''$, $\pi_2 \mapsto^n \pi_2'$ and $\pi_1''\ R\ \pi_2'$; or
   (b) there exists a sequence such that $\pi_1 \mapsto^* \pi_2$;
2. if $\pi_1\ R\ \pi_2$ and if there is no possible transition for π_1 then there is no possible transition for π_2 as well; and
3. if $\pi_1\ R\ \pi_2$, then π_1 is a final state if and only if π_2 is a final state.

Intuitively, a U-simulation is just an ordinary simulation between two systems of transitions (condition 1a), 'until' the LHS transits to the same state in the RHS (condition 1b). We first show that U-simulation implies graph equivalence.

Lemma 6.10.1. *For any states π_1, π_2 and a U-simulation R such that $\pi_1\ R\ \pi_2$, then $\pi_1 \mapsto^* \pi_1^e \iff \pi_2 \mapsto^* \pi_2^e$ for some final states π_1^e, π_2^e.*

Proof. For the direction $\pi_1 \mapsto^* \pi_1^e \implies \pi_2 \mapsto^* \pi_2^e$, it can be proved by induction on the number of transitions in $\pi_1 \mapsto^n \pi_1^e$.

Base case $n = 0$. Since $\pi_1 = \pi_1^e$ and $\pi_1\ R\ \pi_2$, therefore π_2 is a final state.

Inductive case $n = k + 1$. We need to show that if $\pi_1 \mapsto^{k+1} \pi_1^e$ then there must exist a final state π_2^e such that $\pi_2 \mapsto^* \pi_2^e$.

We can factorise the sequence into $\pi_1 \mapsto \pi_1' \mapsto^k \pi_1^e$. By the definition of U-simulation, either (1) there exists an m and two states π_1'', π_2' such that $\pi_1 \mapsto^m \pi_1''$, $\pi_2 \mapsto^m \pi_2'$ and $\pi_1'' \; R \; \pi_2'$; or (2) $\pi_1 \mapsto^* \pi_2$. If it is the former case and since $m > 0$ then by using the induction hypothesis on $k+1-m$, there must exist some state π_2^e such that $\pi_2' \mapsto^* \pi_2^e$. If it is the latter, if π_2 is the final state, it is true. Otherwise, if π_2 is not a final state then we can factorise the sequence into $\pi_1 \mapsto \pi_1' \mapsto^i \pi_2 \mapsto^j \pi_1^e$ where $k = i + j$. Since the machine is deterministic, therefore we have that $\pi_2^e = \pi_1^e$.

For the other direction $\pi_1 \mapsto^* \pi_1^e \impliedby \pi_2 \mapsto^* \pi_2^e$ it can also be proved by induction on the number of transitions in $\pi_2 \mapsto^n \pi_2^e$.

Base case $n = 0$ Since $\pi_2 = \pi_2^e$ and $\pi_1 \; R \; \pi_2$ therefore π_1 is also a final state.

Inductive case $n = k + 1$ We can factorise the sequence into $\pi_2 \mapsto \pi_2' \mapsto^k \pi_2^e$. Since there is a transition for π_2 therefore there must exist a transition for π_1, i.e. $\pi_1 \mapsto \pi_1'$. By the definition of U-simulation, either (1) there exists an m and two states π_1'', π_2' such that $\pi_1 \mapsto^m \pi_1''$, $\pi_2 \mapsto^m \pi_2'$ and $\pi_1'' \; R \; \pi_2'$; or (2) $\pi_1 \mapsto^* \pi_2$. If it is the former, then by using the induction hypothesis, there must exist some state π_1^e such that $\pi_1'' \mapsto^* \pi_1^e$. If it is the latter, then we have $\pi_1 \mapsto^* \pi_2 \mapsto^* \pi_2^e$ because the machine is deterministic. Therefore, $\pi_1^e = \pi_2^e$. □

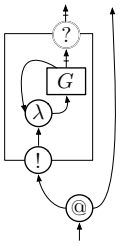
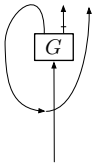
6.10.2 Rewiring

We first consider the equivalence of the graph before and after Beta-rewrite in Prop. 6.10.3.

Definition 6.10.4. We define a relation on states

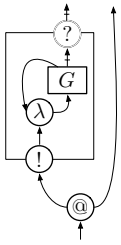
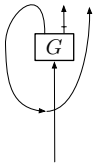
$$(\mathcal{G}[G_1, \ldots, G_n], \zeta_1, \delta_1) \overset{rewire}{\sim} (\mathcal{G}[H_1, \ldots, H_n], \zeta_2, \delta_2)$$

for any n if they satisfy the following conditions:

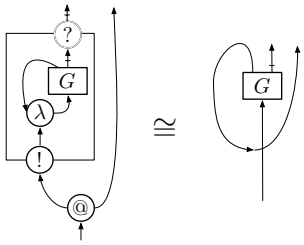
1. for any i, $G_i =$  , $H_i =$  and the o_1-port of the @-node is evaluated;
2. $\zeta_1 = \zeta_2$; and
3. $\delta_1 = \delta_2$ if the token is inside the context graph $\mathcal{G}$.

Lemma 6.10.2. $\overset{rewire}{\sim}$ *is a U-simulation.*

Proof. Suppose the token is inside the context graph $\mathcal{G}$ but not in any of $G_1, \ldots G_n$ and if it has a possible transition on the LHS then the same transition can be applied to the RHS since both sides have the same context graph and token values. The only possible transition that would affect/create more 'holes' would be the δ-rewrite or the μ when the !-box or μ-box in question contains the graph G_i which would create a copy of G_i in the next state. The same δ-rewrite could be applied in the RHS and the two new states would still be in $\overset{rewire}{\sim}$ with $n + 1$ holes.

Otherwise, if the token is at one of G_i, since $G_i =$  , $H_i =$  and the o_1-port of the @-node is evaluated, the token is at the i_0-port of the @-node then it will be sent to the o_1-port. Since o_1 is evaluated, therefore the token must come back and does not change the shape of the graph at all. Then after several transitions and the beta-rewrite the LHS will become the same as RHS and thus satisfying the 1(b) condition of Def. 6.10.3. After the rewrite, the two states will still be in $\overset{rewire}{\sim}$ with $n - 1$ holes. □

Proposition 6.10.3 (Rewiring-equivalence)**.**



if the o_1-port of the @-node is evaluated.

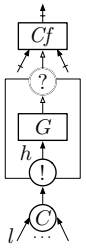
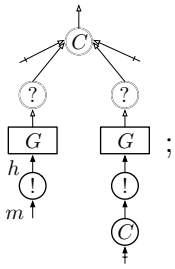
Proof. This is simply the consequence of Lem. 6.10.1 and Lem. 6.10.2. □

6.10.3 Copying

Definition 6.10.5. We define a relation on states

$$(\mathcal{G}[G_1, \ldots, G_n], \zeta_1, \delta_1) \overset{copy}{\sim} (\mathcal{G}[H_1, \ldots, H_n], \zeta_2, \delta_2)$$

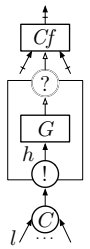
for any n if they satisfy the following conditions:

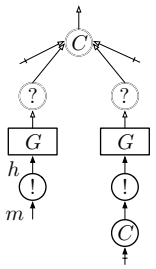
1. for any i, $G_i =$  and $H_i =$  ;

2. $\zeta_1 = \zeta_2$; and
3. $\delta_1 = \delta_2$ if the token is inside the context graph $\mathcal{G}$.

Lemma 6.10.4. $\overset{copy}{\sim}$ *is a one-step simulation.*

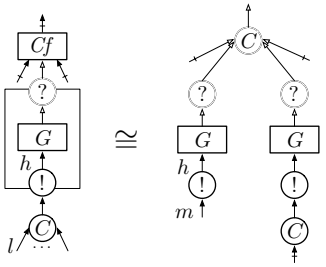
Proof. Suppose the token is inside the context graph $\mathcal{G}$ but not in any of $G_1, \ldots G_n$ and if it has a possible transition on the LHS then the same transition can be applied to the RHS since both sides have the same context graph and token values. The only possible transition that would affect/create more 'holes' would be the δ-rewrite or the μ when the !-box or μ-box in question contains the graph G_i which would create a copy of G_i in the next state. The same δ-rewrite could be applied in the RHS and the two new states would still be in $\overset{copy}{\sim}$ with $n + 1$ holes.

Otherwise, if the token is at one of G_i and one of its inport i_k since $G_i =$ 

, $H_i =$ 

, the token will then be sent to the o_0-port of the !-node and δ-rewrite will happen similar to the proof in Lem. 6.6.5. After the rewrite, the two states will still be in $\overset{copy}{\sim}$ with n holes. □

Proposition 6.10.5 (Copying-equivalence)**.**



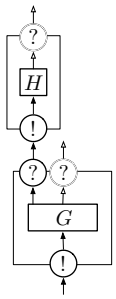
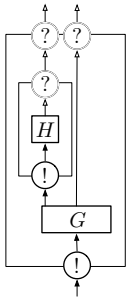
Proof. This is simply the consequence of Lem. 6.6.4 and Lem. 6.10.4. □

6.10.4 Engulfing

Definition 6.10.6. We define a relation on states

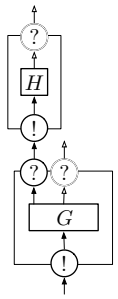
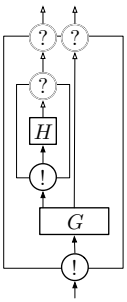
$$(\mathcal{G}[G_1, \ldots, G_n], \zeta_1, \delta_1) \overset{engulf}{\sim} (\mathcal{G}[H_1, \ldots, H_n], \zeta_2, \delta_2)$$

for any n if they satisfy the following conditions:

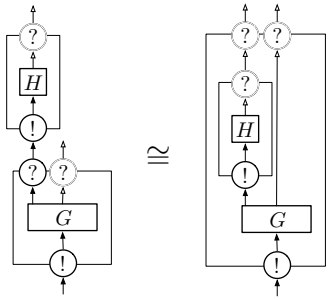
1. for any i, $G_i =$  , $H_i =$  and the !-node is not connected by a C-node nor to a μ-node;
2. $\zeta_1 = \zeta_2$; and
3. $\delta_1 = \delta_2$ if the token is inside the context graph $\mathcal{G}$.

Lemma 6.10.6. $\overset{engulf}{\sim}$ *is a U-simulation.*

Proof. Suppose the token is inside the context graph $\mathcal{G}$ but not in any of $G_1, \ldots G_n$ and if it has a possible transition on the LHS then the same transition can be applied to the RHS since both sides have the same context graph and token values. One possible transition that would affect/create more 'holes' would be the δ-rewrite or the μ when the !-box or μ-box in question contains the graph G_i which would create a copy of G_i in the next state. The same δ-rewrite could be applied in the RHS and the two new states would still be in $\overset{engulf}{\sim}$ with $n+1$ holes. Another possible transition that would affect G_i is the @-rewrite which remove the !-box encapsulating G and thus the two graphs would become the same.

Otherwise, if the token is at one of G_i and since $G_i =$  , $H_i =$  and the !-node is not connected by a C-node nor to a μ-node the token will then be sent to the o_0-port of the !-node and comes back which satisfies the 1(a) condition with $n = 4$. □

Proposition 6.10.7 (Engulfing-equivalence)**.**

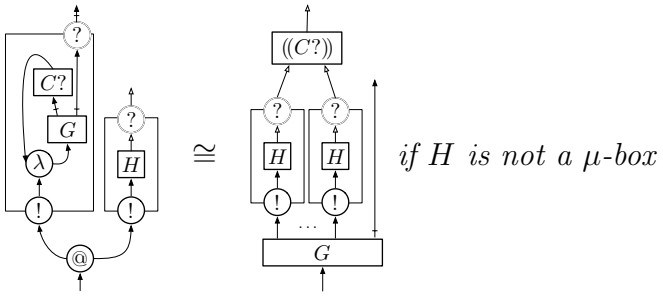


if the !*-node is not connected by a* C*-node nor connected to a* μ*-node*

Proof. This is simply the consequence of Lem. 6.10.1 and Lem. 6.10.6. □

6.10.5 Beta-equivalence

Theorem 6.10.8 (Beta-equivalence)**.**



if H *is not a* μ*-box*

where $C?$ *is a tree of*  *and*  *; and* $((C?))$ *is a tree of*  *and*  *such that* $C?$ *and* $((C?))$ *have the same structure.*

Proof. Since $\cong$ is transitive, therefore this is simply a consequence of Prop. 6.10.3, Prop. 6.10.5 and Prop. 6.10.7. □

6.11 Complexity of transitions

Assuming that a graph is created as a simple recursive data structure, then the following actions are atomic and only cost a constant amount of time: token transitions from one place to another, copying a single node, creating a token and updating a cell value. Therefore, we

can conclude that the time complexity of each transition is either constant or linear to the size of the graph.

Theorem 6.11.1 (Complexity of transitions)**.** *Time complexity of all transitions is either constant or linear in the size of the graph.*

Proof. All the transitions except for the P_a-pass rule, the δ-rewrite, μ-rewrite, Fuse-rewrite, the two transitions triggering and terminating propagation, take only constant time to compute. The P_a-pass rule depends on the number of out-ports of the P_a-node, the δ and μ-rewrite depends on the number of nodes inside the box, the Fuse rewrite depends on the size of the argument graph and finally the two transitions triggering and terminating propagation depend on the number of cells in the graph. Therefore, the time complexity of these transitions are linear to the size of the graph. □

Chapter 7

Experimental Implementations

In this chapter, we will describe different experimental implementations: a native compiler for DGoIM; and OCaml implementations for the dataflow core and the extensions for parameter optimisation. The native compiler is only built for a small subset of the language with only lambda calculus and natural numbers. We did not go further to implement the whole language as its performance is bad compare to existing languages like OCaml. The native compiler and its limitations will be described in section 7.1. The main contributions however are the OCaml implementations for the whole language which will be described in section 7.2 and section 7.3 which will discuss the implementation of the dataflow core and the extension for parameters optimisation respectively.

7.1 Native compiler for DGoIM

The native compiler transforms a term in lambda calculus to x86 assembly codes. It is meant to follow the specification of DGoIM where the nodes are translated into machine instructions to guide the movement of the token and any possible rewrites. The position of the token is represented by the program counter, indicating the current address of the code being executed. A pass transition is simply a jump instruction which modify the program counter to the destination. The rewrite flag is a value in a register, and the computation stack

```
type var = string
type term t =
    Const of int
  | Var of var
  | Abs of var * term
  | App of term * term
  | Add of term * term
```

Figure 7.1: Syntax tree

is simply the stack in the main memory. As a result, pass transitions can be seen as jumps while rewrite transitions can be implemented as rewrites of the assembly codes, for example, evaluating $1 + 1$ rewrites the code of $+$ to 2; and copying a box in DGoIM is the same as copying a chunk of memory in the code space. Therefore, the code is modified in runtime. The code that can rewrite itself during execution is often referred to as *self-modifying code* where instructions loaded into the code space are modified along their execution. There might be more traditional ways to compile this language without the needs to modify the instructions, but we choose to implement in this way as an experiment.

As mentioned before, we only work with a small set of syntax, lambda calculus with numerals, but not the whole language because of performance issues. The syntax tree is shown in Fig. 7.1.

7.1.1 Modified abstract machine

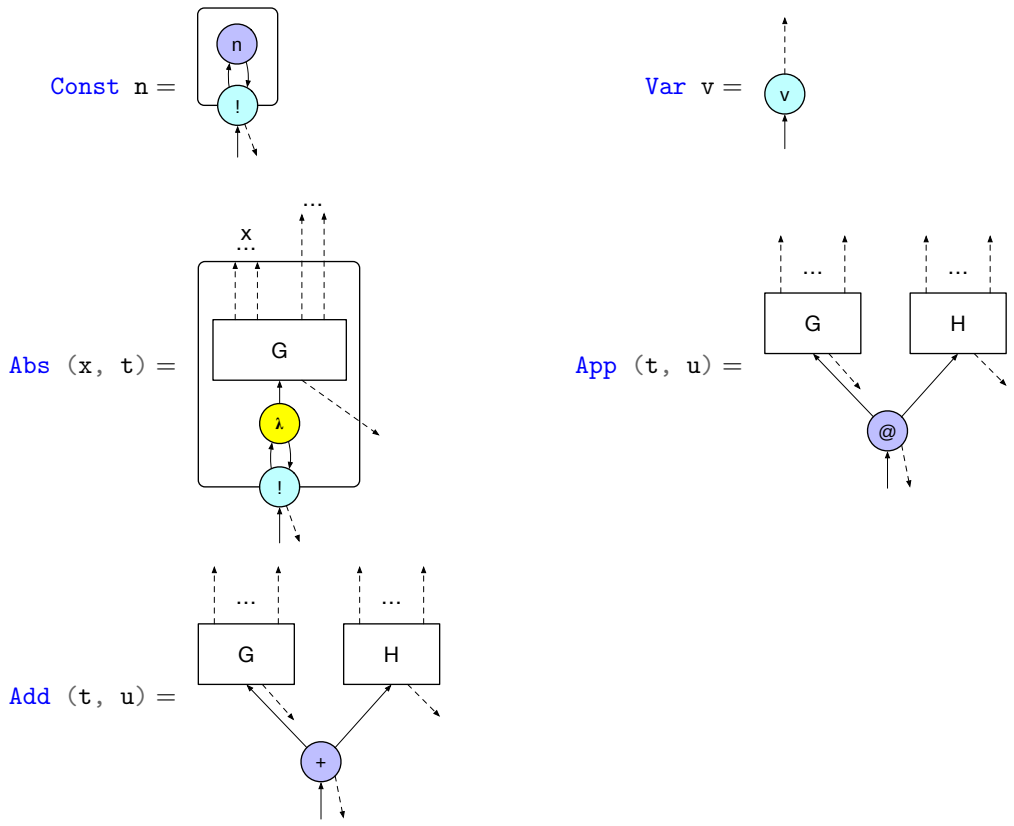


Figure 7.2: Translation to graphs

The graph abstract machine used in the implementation is slightly different from the DGoIM specifications in several ways. The (informal) translation is shown in Fig. 7.2. Every port is now separated into two parts, one for upward direction and one for downward. For example, in the translation of `Const n`, the n-node has two ports, `in_up` and `in_dn` in which the token comes in from `in_up` and leaves from `in_dn`. As a result, the token no longer needs to remember its direction. The dotted line represents a possible *computed jump* in the x86-code, where the address is only known at runtime. Therefore, these lines are not going to be connected to any nodes right after the translation. This also means that all the return addresses are only known at runtime, and for the sake of convenience they will be omitted

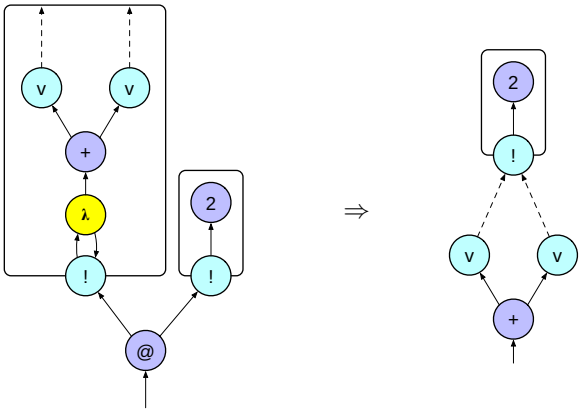


Figure 7.4: Beta-rewiring in native compile

in any upcoming diagrams.

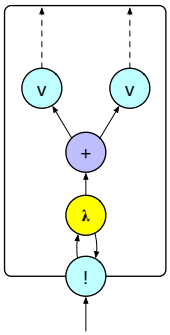


Figure 7.3: $\lambda x.x + x$

Furthermore, there is no need for having contraction nodes because every variable simply points towards the same node (having the same jump target), so instead each variable instance is translated to a var-node. For example, the term $\lambda x.x + x$ will be transformed (informally) into Fig. 7.3. Also, because of that, the box-stack is no longer needed, instead it is replaced by a stack storing return addresses. When the token comes back to the `in_dn`-port, it will look up at the top of the address-stack and jump to that node.

In addition, the beta-rewiring rule will instead connect all the var-nodes to the argument as shown in Fig. 7.4. Intuitively, the implementation of the beta-rule consists of two steps: 1) we change the entry of the @-node to a jump to the +-node; and 2) we change the jump target of the two v-node to the !-node. In this way, whenever the @-node is visited again,

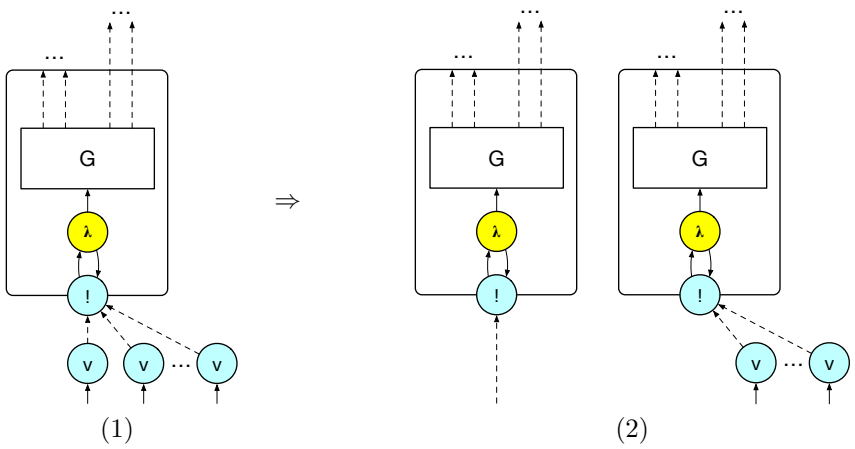


Figure 7.5: Copying in native compile

it will be directed to the +-node. The copying rule is initiated by the var-node as shown in Fig. 7.5. Lastly, we do not implement copying for constants.

7.1.2 Code generation

The code generation is defined as a function

```
term_to_code : term -> label * code * auxs
```

where `label` is the in-up-port of the translated sub-graph and `auxs = (var * label) map` is a mapping of variable names to labels indicating the out-up-ports.

To make an example, we will only show the code generation for `Var v`. The full implementation can be found here[1]. The code is shown in Fig. 7.6. Line 3-4 create the labels of the var-node, which is useful for locating relative jumps and also for human readability. The actual instructions of the var-node begin in line 6 where the machine first tests if this var-node has been optimised or not, which needs a bit more explanation. Under the modified abstract machine, the var-node is never going to be visited unless it has been involved in a beta-rewrite before which has changed line 29 to `movabs X rax` where `X` is the address of the

[1] `https://github.com/cwtsteven/DGoIM-Compiler`

```
let rec term_to_code term =
match term with
  | Var v ->  let var = "var_" ^ string_of_int (!count) in
              let _ = count := !count + 1 in

              let code =
                var^"_in_up:\n"
                  (* test if this var-node has been rewired or not *)
              ^  mov "$1" rax
              ^  cmp "$0" rax
              ^  je (var^"_out_up_jmp") (* jump to the rewired target *)

                 (* test if previous node is var *)
              ^  lookup 0 rax
              ^  cmp s.var rax
              ^  je (var^"_out_up")
                 (* internal jump *)
              ^ var^"_not_from_var:\n"
                 (* computed jump instructions address *)
              ^  lea (var^"_out_up_jmp") rip rax
              ^  push rax
                 (* entry address*)
              ^  lea (var^"_in_up") rip rax
              ^  push rax
              ^  push s.var
              ^ var^"_out_up:\n"
              ^ var^"_out_up_jmp:\n"
                 (* should have been rewritten by a beta-rule before *)
              ^  movabs "$0" rax
              ^  jmp_reg rax
              ^  ud2
              in
              (var^"_in_up", code,
                let vtbl = Hashtbl.create 0
                in Hashtbl.add vtbl v [var^"_out_up_jmp"]; vtbl)
```

Figure 7.6: Code generation (Var)

argument in the beta-rewrite indicating the jump target. For example, evaluating the term

$$\texttt{let } f = \lambda x.t \texttt{ in let } g = f \texttt{ in } g\ 1$$

will bind $\lambda x.t$ to two consecutive var-nodes, representing f and g. Every time when g is accessed, the token traverse through the chain of var-nodes to get to the actual code of the lambda abstraction. The idea of the optimisation is to rewrite the jump target ("$0" in Line 29) of the first var-node to the address of the actual value; and the flag in line 9 from "$1" to "$0" after the first visit. This part of the code however is not shown here but can be found in the repo[2].

Continuing from line 10, if the var-node has already been optimised then the token will jump directly to `var_out_up_jmp` in line 27 which stores the instructions for the computed jump. The line `movabs X rax` moves the absolute address `X` into the register `rax` where `X` was initially `$0` but should have been rewritten by beta-rewiring to some address before.

Going back to line 10, if this var-node has not been optimised yet then the machine first checks if the previous node is a var-node, if yes that means the token is in the middle of the var-nodes chain and as such it does not need to do anything and simply jumps to the computed jump instructions. However, if the previous node is not a var-node then it needs to remember the entry address and the computed jump instructions address so that when the token finally hits an abs-node, Line 9 and Line 29 can be modified.

Finally, it returns a triplet containing the entry address `var_in_up`, the instructions `code` and a mapping that maps the variable `v` to a singleton list `[var_out_up]`. The list is used in the code generation of `Abs (x,t)` in order to rewrite the address `X` in the line `movabs X rax`. The code can be found in the same repo[2].

[2] https://github.com/cwtsteven/DGoIM-Compiler/blob/master/src/backend/CodeGeneration.ml

7.1.3 Evaluation

```
let f0 = (fun x -> x + 1) in
⋮
let fn = (fun x -> fn-1 (fn-1 x)) in
⋮
fn 1
```

Figure 7.7: Testing program

As expected, the performance of the code generated is lacking when the same function is called many times. We compare the performance of the native compiler and OCaml by executing the testing program shown in Fig. 7.7 in which 2^N copies of the function `f0` will be created. As shown in the Tab. 7.1, the execution time of the testing program in native compiler is exponential while that in OCaml is relatively constant.

N	native compiler	OCaml
10	0.012s	0.001s
15	0.315s	0.001s
20	9.553s	0.001s
21	19.89s	0.002s
22	33.87s	0.002s
23	40.37s	0.002s

Table 7.1: Native compiler vs OCaml

One of the reasons is that self-modifying code in general can be problematic and slow under modern processor architecture. For example, most CPUs have one or more caches to reduce the time for fetching instructions from the main memory. By using this optimisation, a page of instructions is loaded into the caches such that the CPU does not need to fetch each instruction one-by-one. However, as self-modifying code will rewrite itself during execution, the changes might not be visible to the caches and thus incorrect instructions might be executed. Most processors deal with this situation by flushing the caches when their contents are violated. As code rewrite happens a lot in the generated code of the native compiler, its performance is also highly affected by it.

Secondly, the generated code contains many *computed* jumps, which are problematic with

instruction pipelines. For static or relative jumps, the CPU knows where to find the next instruction in order to continue with the pipeline. For computed jumps, it is only possible if dynamic branch target prediction is available in the processor. Another possible improvement is to use direct far jumps instead of computed jumps, as the target addresses are only computed once. However, direct far jumps are currently invalid in the AMD x86-64 instruction set [49].

Given the several major caveats of the native compiler, we decided to not go further and implement the TSD language on top of it. Instead, it is implemented as a library module in OCaml, which will be discussed in the next sections.

7.2 The dataflow core as an OCaml library

The implementation of the TSD language is separated into two parts: (i) the dataflow core and (ii) the extensions for parameter optimisations. In this section, we will describe the former.

We implement the dataflow core as a library module for OCaml, together with a PPX language extension[3]. The library serves as an embedded DSL, which is blended into the host language more seamlessly via the language extension. The implementation is facilitated by OCaml's imperative features and its facilities for type-casting. The type of dataflow graph is implemented as a datatype in OCaml as follows:

```
type 'a graph =
    Thunk of (unit -> 'a)
  | IF_Thunk of (unit -> 'a graph)
  | Cell of ('a * 'a graph) ref
```

It can either be a dependency `Thunk`, a `Cell`, or an if-then-else graph `IF_Thunk`. The cell is implemented as a reference storing the current value and a thunk representing the dataflow graph. The if-then-else graph captures the semi-strict behaviour of the conditionals in the

[3] https://github.com/cwtsteven/TSD

calculus where both the branches are evaluated. Note that the programmer does not explicitly use these constructors to create dataflow graphs, but instead the PPX extension has taken care of that, which will be described later in this section. Functions `peek, step, link, assign, root, ref` work as described earlier where `ref` is replaced by `cell`. The types of these operations are defined as follows:

```
val peek : 'a graph -> 'a
val root : 'a graph -> 'a graph
val cell : 'a graph -> 'a graph
val link : 'a graph -> 'a graph -> unit
val (<~) : 'a graph -> 'a graph -> unit (* infix for link *)
val assign : 'a graph -> 'a -> unit
val (<:=) : 'a graph -> 'a -> unit (* infix for assign *)
val set : 'a graph -> 'a -> unit
val (<:~) : 'a graph -> 'a -> unit (* infix for set *)
val step : unit -> bool
```

The `set` operation is a combination of `assign` and `link`. The type of `link` and `assign` are different from the calculus, as one would expect their first arguments to have a `cell` type. However, we decided to not implement the cell type, so a cell is also of type `'a graph`, which allows us to remove the `deref` operation in order to keep the syntax succinct. As a result, the difference between a cell and a non-cell graph is not known at compile time, which could lead to runtime error when one trying to use `link` on these graphs. It might be possible to implement the `cell` type, and let the compiler add the implicit `deref` operation at the right places, but that is left for future improvements. Operators related to parameter optimisation will be introduced in the next section.

Internally, the whole dataflow graph is maintained using a heterogeneous list of all cells. The `Obj` module is used to overcome OCaml's type restrictions. However, the internal details are inaccessible to the programmer, who may use the module only via the PPX extension `[%dfg t]` which *translates* a pure term `t` into a dataflow graph. By pure term, we mean that t does not contain any `cell`, `link`, `assign`, `root`, `peek` nor `step` operator. These functions are not translated and should only be used outside the `[%dfg t]` tag. The translation $(\ulcorner - \urcorner_V)$ is indexed by a set V of variables bounded in the `[%dfg t]` tag:

$$\ulcorner x \urcorner_V = \texttt{x} \quad (\texttt{if } x \notin V)$$

$$\ulcorner x \urcorner_V = \texttt{lift x} \quad (\texttt{if } x \in V)$$

$$\ulcorner c \urcorner_V = \texttt{lift c}$$

$$\ulcorner \lambda x.t \urcorner_V = \texttt{Thunk ((fun () -> (fun x -> peek } \ulcorner t \urcorner_{V \cup \{x\}} \texttt{), empty))}$$

$$\ulcorner t\ u \urcorner_V = \texttt{apply } \ulcorner t \urcorner_V \ \ulcorner u \urcorner_V$$

$$\ulcorner \texttt{if } b \texttt{ then } t \texttt{ else } u \urcorner_V = \texttt{ite } \ulcorner b \urcorner_V \texttt{ (fun () -> } \ulcorner t \urcorner_V \texttt{) (fun () -> } \ulcorner u \urcorner_V \texttt{)}$$

where `lift` creates a graph from ordinary OCaml term, `apply` simulates the application in the TSD calculus and `ite` implements the semi-strict behaviour of branching. Their implementations are as follows:

```
let lift t = Thunk (fun () -> t)
let apply t u = Thunk (fun () -> (peek t) (peek u))
let ite b t1 t2 =
  let t1 = t1() in let t2 = t2() in
  IF_Thunk (fun () -> if peek b then t1 else t2)
```

A simple example written in both TSD term and the OCaml implementation is shown in Fig. 7.8.

$\texttt{let } f = \lambda x.\lambda y.x * y \texttt{ in}$
$\texttt{let } x = \texttt{ref } 1 \texttt{ in}$
$\texttt{let } y = \texttt{ref } 2 \texttt{ in}$
$\texttt{let } m = f\ (\texttt{deref } x)(\texttt{deref } y) \texttt{ in}$
$\texttt{link } x\ (y * 1)$

(a) A TSD term

```
let ( * ) = lift ( * );;
let f = [%dfg fun x y -> x * y];;
let x = cell [%dfg 1] ;;
let y = cell [%dfg 2] ;;
let m = [%dfg f x y];;
link x [%dfg y * 1];;
```

(b) OCaml program

Figure 7.8: Example program in the OCaml implementation

Notice that recursion is not translated into OCaml, but we can still use the OCaml recursion to create a number of cells recursively. For example, the following function creates n cells and put them into a list.

```
let rec create n =
  match n with
  | 0 -> []
  | n -> cell [%dfg 1] :: create (n - 1)
```

As shown in the above examples, free variables inside `[%dfg]` tag are expected to be already "lifted" to graphs. For example, the `*` operator needs to be manually lifted before it is used inside the tag. This OCaml implementation does not implement the transparent lifting of dataflow graphs. A full implementation based directly on the abstract machine is a more substantial endeavour, which we leave as further work.

7.2.1 Incremental version

Right now, every cell is recomputed during a `step`, which could be inefficient for systems that only have small changes overall. In light of this, we implemented an incremental version which re-evaluates only those cells that need to be recomputed in a step. Instead of recomputing all the cells, we keep a list of *dirty* cells and only recompute them during a step. This list is maintained by all the operations, for example when a cell is being `link`ed, it will be put into the dirty list and after a `step`, all the cells that depend on this cell will be added to the list. Therefore, we also need to maintain a list of dependant cells for each cell which generates a bit more overheads than the original version.

7.2.2 Benchmarks

We tested two versions of the code, with and without incrementalisation. We ran similar (as similar as possible) versions of the same program against (1) Reactive ML, (2) Zélus, and (3) JaneStreet Incremental library. We chose the programs from the documentations of those programming languages, presumably as representative of their strengths. For each example, we could vary the size of the graph for up to 10^7 nodes. All times expressed in milliseconds.

against RML and Zélus The benchmarks are shown in Table. 7.2. We have chosen four examples to test against Zélus: (a) the alternating-sum automata from the introduction, (b) heater simulation, (c) light controller and (d) pendulum simulation. These examples can also be found on the Zélus website. For RML we have: (e) the trajectory of planets which is an n-body simulation, (f) FIR filter, (g) IIR filter, (h) page rank algorithm, (i) map function acting on large dataflow graph and (j-m) folding on a large dataflow graph.

ReactiveML is intended for concurrent programming and thus processes are created in different threads internally. Therefore, in some cases it will have more overheads when compared to the TSD implementation. In most cases RML and TSD both have similar performances except for the FIR, IIR filters, fold-max and fold-sum. For FIR and IIR filters, it is expected that all cells are going to be recomputed at each `step` which hurts the performance of inc. TSD as it produces a lot of overheads to maintain the list of dirty cells. For the fold-max and fold-sum examples, we can see a huge difference in TSD compared to their second example (labelled as 2). Fig. 7.9 shows the differences in the two fold-sum programs where `ins` is a list of cells all initiated with constant 1 and `stabilise` is a function which performs `step`s until there is no more changes to the graph. In fold-sum, cells are created for each intermediate result when an input cell is added to the accumulator. This creates a long chain of cells which when `stabilise()` is called, leads to the changes having to go through all the intermediate cells. What is worse is that in TSD, all cells are recomputed regardless of whether they need to be. On the other hand, fold-sum 2 does not create any intermediate cells which means when `stabilise()` is called, only the input cells are recomputed and the fold result is calculated afterwards. This example shows that it is not always the case that adding cells to the graph will improve performance.

TSD also has similar performance as Zélus which is encouraging as Zélus is a very similar language.

size	TSD	inc. TSD	RML	Zélus
10^5	14	15	33	38
10^6	136	146	306	92
10^7	835	1,463	2,786	776

(a) alternating-sum automata

size	TSD	inc. TSD	Zélus
10^4	1	1	28
10^5	1	2	49
10^6	2	1	234

(b) heater simulation

size	TSD	inc. TSD	RML	Zélus
10^5	12	14	71	30
10^6	140	138	705	89
10^7	1,301	1,292	6,466	648

(c) light controller

size	TSD	inc. TSD	Zélus
10^5	85	191	88
10^6	861	1,887	638
10^7	8,793	21,221	6,464

(d) pendulum

size	TSD	inc. TSD	RML
10	6	6	7
10^2	323	336	597
10^3	46,644	47,698	54,268

(e) planets

size	TSD	inc. TSD	RML
10^2	32	226	32
10^3	381	3,905	272
10^4	11,630	46,323	7,728

(f) FIR filter

size	TSD	inc. TSD	RML
10	116	431	67
10^2	1,243	8,351	468
10^3	17,765	198,376	6,465

(g) IIR filter

size	TSD	inc. TSD	RML
10	1	1	2
10^2	10	2	11
10^3	2,970	387	3,060

(h) page rank

size	TSD	inc. TSD	RML
10^3	2	1	2
10^4	34	14	2
10^5	462	188	23

(i) map

size	TSD	inc. TSD	RML
10^2	1	2	2
10^3	170	2	1
10^4	5,381	32	2

(j) fold-max

size	TSD	inc. TSD	RML
10^2	1	2	2
10^3	2	2	2
10^4	13	26	2

(k) fold-max 2

size	TSD	inc. TSD	RML
10^2	2	2	2
10^3	141	5	1
10^4	52,868	51	2

(l) fold-sum

size	TSD	inc. TSD	RML
10^2	2	1	2
10^3	2	2	2
10^4	10	26	2

(m) fold-sum 2

Table 7.2: Benchmarks against RML and Zélus

```
let out = fold_left (fun acc i -> cell [%dfg i +^ acc]) (lift 0) ins in
List.map (fun i -> set i 2) ins;
stabilise ();
```

(a) fold-sum

```
let out = fold_left (fun acc i -> [%dfg i +^ acc]) (lift 0) ins in
List.map (fun i -> set i 2) ins;
stabilise ();
```

(b) fold-sum 2

Figure 7.9: Difference between fold-sum and fold-sum 2

size	inc. TSD	JS Incr.
10^3	3	5
10^4	11	15
10^5	167	52
10^6	1,796	733
10^7	2,243	8,222

(a) chain of propagation

size	TSD	inc. TSD	JS Incr.
10^3	2	2	5
10^4	13	2	7
10^5	121	5	15
10^6	1,629	28	100
10^7	15,510	275	998

(b) tree propagation

size	TSD	inc. TSD	JS Incr.
10^3	2	1	7
10^4	8	3	7
10^5	152	12	20
10^6	1,683	48	132
10^7	17,217	384	1,172

(c) field of propagation

Table 7.3: Benchmarks against JS incremental

against JS incremental The benchmark tables are shown in Table. 7.3. There are three examples: (a) a long-and-thin chain of propagation, (b) a short-and-wide chain of propagation and (c) a tree-like chain of propagation. JS incremental library is a highly-tuned OCaml library used in production by JaneStreet. It does not support cyclic dependencies, so most examples could not be implemented in it. The JS incremental library not only implements the propagation of changes, but also has memoization that remembers when the result of some function calls. However, in the three benchmarks, memoization does not help because the recomputation of the nodes require new arguments. This is probably the reason why the JS incremental library is slight slower than inc. TSD. As for the plain TSD, it performs

badly in these programs because many cells that are recomputed do not actually need to be recomputed.

We can conclude that even as a slightly-optimised proof-of-concept implementation, the performance of the dataflow core is satisfactory.

7.3 Extensions for parameters optimisation

The implementation of the extensions[4] is based on the plain version of the dataflow core for its simplicity and also because in systems optimisation, it is often the case that all the cells would need to be recomputed. Therefore, having the overhead of maintaining the dataflow graph would lower the performance and thus unnecessary.

The extension requires several additions to the implementation. First of all, names are represented as types and thus named vector types *Vec a* are implemented as polymorphic types `'n sym = 'n * vec` where `vec` is a dictionary of floats. Unlike in the TSD calculus in which only a restricted form of polymorphism is provided, we exploit the type system of OCaml to provide a full ML-style name-polymorphism which is stronger than the syntactic polymorphism. As discussed before, we believe that this is possible in the TSD calculus, but requires some extensive changes to the graph abstract machine. These possible changes will be discussed in section 8.3.

```
type vec = float Dictionary.t
type 'n sym = 'n * vec
```

Secondly, the data type `graph` needs to be modified to handle parameters and the fusion binder:

```
type _ graph =
  | Thunk : (vec -> 'a) * vec -> 'a graph
  | IF_Thunk : (vec -> 'a graph) * vec -> 'a graph
  | Cell : ('a * 'a graph) ref
  | Params : 'n * vec -> 'n sym graph
```

[4] https://github.com/cwtsteven/Transparent-Synchronous-Dataflow

Compare to the dataflow core, `Thunk` now consist of a function `vec -> 'a` that takes a dictionary and returns `'a`; and a dictionary. The function is basically the skeleton function after fusion is performed, and the dictionary is the collection of parameters. This means that we do not perform `fusion` in runtime, but instead the decoupling is done in compile time where we simply collect the parameters while traversing the abstract syntax tree. For example, the following code

```
let m = [%dfg x + pc 1 + pc 2] in
let n = [%dfg m + pc 3] in
```

is translated into

```
let m = Thunk ( (fun ps -> x + !(find 0 ps), (find 1 ps))), [(0, 1), (1, 2)] )
let n = Thunk ( (fun ps -> (_peek m ps) + (find 2 ps)), [(0, 1), (1, 2), (2, 3)] ) in
```

The major challenge is to implement the generative typings of the fusion binder, where each binder should have its own unique name. The binder is implemented as a language extension. Its usage is as follows:

```
let%fuse (f, ps) = n
```

where

```
f : c sym graph -> a graph
ps : c sym graph
```

Therefore generating a new unique type is the same as generating a new `c` for each binder, and this is accomplished by using GADT to define natural numbers as types.

```
type tsd_z = TSD_Z
type 'n tsd_s = TSD_S : 'n -> 'n tsd_s
```

The type `z` represents zero and `'n s` is the type-level successor operator. The fusion binder is then translated to

```
let (f, ps) = _fusion n !paramtype
```

where

```
val _fusion : 'a graph -> 'n -> ('n sym graph -> 'a graph) * 'n sym graph
```

while `paramtype` is a reference containing the current index. Every time when the compiler encounters a fusion binder, the index `paramtype` will 'increase' and thus gives a unique vector type. Earlier in this section we have seen that the variable `n` is a `Thunk` consisting of a skeleton function and a dictionary of parameters. The `_fusion` function take advantage of it and is defined as follows:

```
let _fusion t n =
  match t with
  | Thunk (f, ps) ->
      (fun vecg -> Thunk ((fun vec -> f vec), peek vecg)),
      Params (n, ps)
  | (* other cases *)
```

where `Params (n, ps)` has type `n sym graph` which is defined in the `'a graph` data type, representing parameters inside the domain specific language. After fusion, we can use the function and the parameters as follows to create a new graph that depends on a different set of parameters:

```
let%fuse (f, ps) = n in
let (|+|) = lift (|+|) in
let n' = f [%dfg ps |+| ps] in
...
```

where the operators {⊞,⊠,⊡,`fold`} are implemented with the following types:

```
val plus : 'n sym -> 'n sym -> 'n sym
val (|+|) : 'n sym -> 'n sym -> 'n sym (* infix for plus *)
val mult : float -> 'n sym -> 'n sym
val (|*|) : float -> 'n sym -> 'n sym (* infix for mult *)
val dot : 'n sym -> 'n sym -> float
val (|.|) : 'n sym -> 'n sym -> float (* infix for dot *)
val fold : ('n sym graph -> 'n sym graph) -> 'n sym graph -> 'n sym graph
```

Finally, in the experimental implementation we did not implement the name checking in the fusion-rule where the return type cannot contain its bound name, i.e. $a\#\tau$. Such checking is possible, but requires modifications to the type checking procedure in the compiler.

Notice that we do not perform any benchmarks against other deep learning frameworks because of several reasons. Firstly, most frameworks provide automatic differentiation, which is much faster than numerical differentiation. Secondly, it is simply a proof-of-concept implementation without many optimisations. Other frameworks usually employ specialised numerical computing frameworks such as NumPy, which are much faster than a naive implementation that uses lists or dictionaries. As these are the key elements that have the strongest effect on the performance, comparing TSD with other frameworks at this point will not bring any insights.

Chapter 8

Evaluation

In the beginning of this dissertation, we set out to design a language that (1) has the ability to create straight-line code easily; (2) provides a safe way to manipulate arrays; and (3) is built on top of a unified semantics harmonising lambda calculus and dataflow graphs. In this chapter, we will review and discuss these objectives, and compare TSD with other similar languages.

8.1 Straight-line code construction

As mentioned before, many conventional automatic differentiation methods work on straight-line codes, which are basically ground-type pure functional recursion-free codes. In TSD, straight-line codes are created in the form of dataflow graphs that exhibit these properties, as shown in Def. 6.2.3. These graphs are created using imperative commands, and the resulting graphs are guaranteed to be *pure* and *recursion-free*. For example, the code

$$\texttt{let}\ x = \texttt{ref}\ 1\ \texttt{in}\ \texttt{let}\ y = \texttt{ref}\ (f\ x)\ \texttt{in}\ \ldots$$

first creates a cell `x` initialised with `1` and executes all the commands in `f` with respect to the cell `x` but leaves a pure dataflow graph behind no matter what is inside `f`. Larger graphs

can also be created using recursion, for example the function

$$\texttt{let } sum = \lambda x.\texttt{rec } f.\lambda n.\texttt{if } n <= 0 \texttt{ then } 0 \texttt{ else } x + f\ (n-1) \texttt{ in} \ldots$$

creates a graph that adds the graph `x` to itself `n` times.

On top of that, dataflow graphs in TSD are also

modal - they are created using imperative commands, while changes in the inputs are propagated through the graph using `step`;

eager - they are constructed on-the-fly while being executed at runtime, for example the code

$$\texttt{let } y = \texttt{ref } (f\ x) \texttt{ in } \ldots$$

executes the term $f\ x$ with input data of x while it constructs the dataflow graph for y;

dynamic - their shape can be modified manually during execution in between `step`s as shown in the example program in section 4.5. However, if one uses fusion to create copies of the graph, there is no reference to the intermediate cells for the programmer to change the cells' dependencies. For example, in the code

```
let x = ref 0 in
let y = ref (deref x)  in
let m = ref (deref y * pc 1 + pc 0)) in
fuse (f, ps) from m in
let n = f ps in
link y (deref x + deref x);
```

the `link` operation will only change the shape of `m` but not `n`. In other words, the skeleton function f can only create static graph models. In order to create dynamic models, the programmer has to fuse the model again after changes are made. For example,

```
let x = ref 0 in
let y = ref (deref x)  in
let m = ref (deref y * pc 1 + pc 0)) in
fuse (f, ps) from m in
let grad = num_grad f ps mesloss 0.01 in
let optimise = gradient_descent ps grad 0.1 in
optimise ...;
(* modify the shape of the graph *)
```

```
9  link y (deref x + deref x);
10 (* fuse again after changing the graph *)
11 fuse (f, ps) from m in
12 let grad = num_grad f ps mesloss 0.01 in
13 let optimise = gradient_descent ps grad 0.1 in
14 optimise ...;
```

This could be inefficient if for example the graph is different for each data sample because a new gradients graph is created every time. A possible way to reconcile this problem is to export the reference of intermediate cells such that when `link y ...` is executed, all the copies are also modified. But this requires extensive modifications to the abstract machine and is left for future development.

efficient - they can be seen as partially evaluated, as the machine eagerly reduces those expressions that can be reduced. For example, the code

$$\texttt{let}\ x = \texttt{ref}\ 1\ \texttt{in}\ x + (1 + 1)$$

will leave a resulting graph corresponding to `x + 2` which is a smaller graph that runs faster;

parallel - they can be executed in parallel in which the programmer has fine-grained controls over both operation and pipeline parallelism as shown in section 4.1; and

transparent - they are nothing more special than a generalised constant value. The distinction between them is transparent to programmers and is dealt with by the abstract machine. The primitive operations are used as if they were applied to constant values, for example the `if` expression is strict when the condition is a graph (both branches are evaluated) but is non-strict when the condition is a normal value.

8.2 Symmetric optimisation

The second objective of this thesis is to address the problem of dimensionality when optimising model parameters using array programming. Our solution employs generative typing for every fusion such that the fused parameters are specific to a particular model and only that

model can subsequently update them. The language is also packaged with a set of symmetric operations in order to restrict the programmer from accessing individual elements of the parameters and to express optimisation algorithms such as gradient descent only using these operations. They include vector addition, scalar multiplication, dot product and folding over basis (`foldp`). It is easy to see that the first three are indeed symmetric. As for `foldp` it is only symmetric under certain conditions. The folding operation has type

$$\texttt{foldp} : (\mathit{Vec}\ a \to \mathit{Vec}\ a) \to \mathit{Prm}\ a \to \mathit{Vec}\ a$$

and we have proved that if the function `f` and all its referenced variables with function type are *pure* then `foldp` f ps is symmetric (section 6.9).

However, certain situations require that f is not pure but still maintains symmetry. For example, in the linear regression program in section 4.1, when trying to put cells into the gradient graph we modified the mean squared error function as follows:

```
1 let mse' = λy.λy_exp. ref ((y - y_exp) * (y - y_exp)) in ...
```

This function is then passed to `num_grad` which is then used by the `foldp` operation. In this particular example it is still symmetric because the effects of creating a cell is *local* to every dimension. Therefore, a more sophisticated theorem concerning local effects is needed.

Furthermore, the symmetry property is currently not guaranteed by the type system, but one can imagine it is not difficult to devise a static analysis to check if the function `f` would induce any effects when executed. This is commonly known as a *purity analysis*. For instance, one possible way is to inline every variable with function type in `f` and then syntactically checks if `f` contains any impure operations.

Finally, we have shown that optimising algorithms can be written using only the operations in TSD, in particular gradient descent with numerical differentiation. The example program is shown in section 4.1.

8.3 A unified semantics

The last objective is to formalise the language using a unified semantics harmonising the interaction between lambda calculus and dataflow graphs in order to ensure safety and theoretical efficiency. In this thesis, we decided to build the TSD language on top of a graph abstract machine inspired by the Dynamic Geometry of Interaction Machine (DGoIM). Under this framework, a program is represented by a graph similar to a syntax tree and its execution is given by a token traversing the graph while collecting information and guiding how the graph should be modified. As shown in the specification of TSD in chapter 5, the interaction between computation graphs (the graph that corresponds to a program for evaluation) and dataflow graphs is seamlessly brought together under the same framework. Dataflow graph is nothing more special than a generalised constant in the environment that is shared by the rest of the computation graph.

By using this formalism we have proved the following theorems for the TSD language:

determinism - we show that the parallel evaluation of the dataflow graph produces deterministic results in section 6.5;

safety - we show that for any well-typed program the abstract machine executes without reaching illegal states in section 6.8

termination - we show that the execution of any recursion-free program with dataflow terminates by reaching the final state of the abstract machine in section 6.8;

symmetry - we show that the result of any program (under certain conditions) is independent of how parameters are collected in section 6.9;

equivalences - we show that beta-equivalence for usual call-by-value languages hold in our language in section 6.10; and

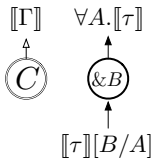
efficiency - we show that no operation is 'expensive' in terms of time and memory utilisation in section 6.11.

Lastly, this style of semantics also allows us to create visualisers easily that visualise both

the process of evaluating computation graph and executing dataflow graphs.

Name polymorphism As mentioned in section 3.3, name-polymorphism is provided syntactically in TSD. Polymorphic functions are copied and renamed such that they are type checked in another simply-typed intermediate language. As discussed in the same section this restricts the way of how polymorphic functions can be defined. Furthermore, when using modules or libraries their source code will have to be compiled all together. We believe that a Hindley-Miller style name-polymorphism is possible and needed for future development.

In order to accommodate it, one possible way is to introduce names into the graphs. For instance, the port of every nodes will have to contain the universal quantifiers over names. A renaming node is also required to renaming sub-graphs. If we translate directly from the TSD calculus to graphs then the variable rule would be as follow:



where $\&B$ is the renaming node and $[B/A]$ denotes the substitution of the set of names A with B. When executed all the names in A in the reachable graph from the $\&B$-node will be changed to the corresponding name in B except for those names that are reachable by another $\&Z$-node. There would also require rewrite transitions involving $\&B - C$-connection where the $\&B$-node is distributed to every in-port of the C-node.

8.4 Comparisons with other languages

In this section, we compare TSD with similar languages in deep learning and reactive programming.

8.4.1 Deep learning frameworks

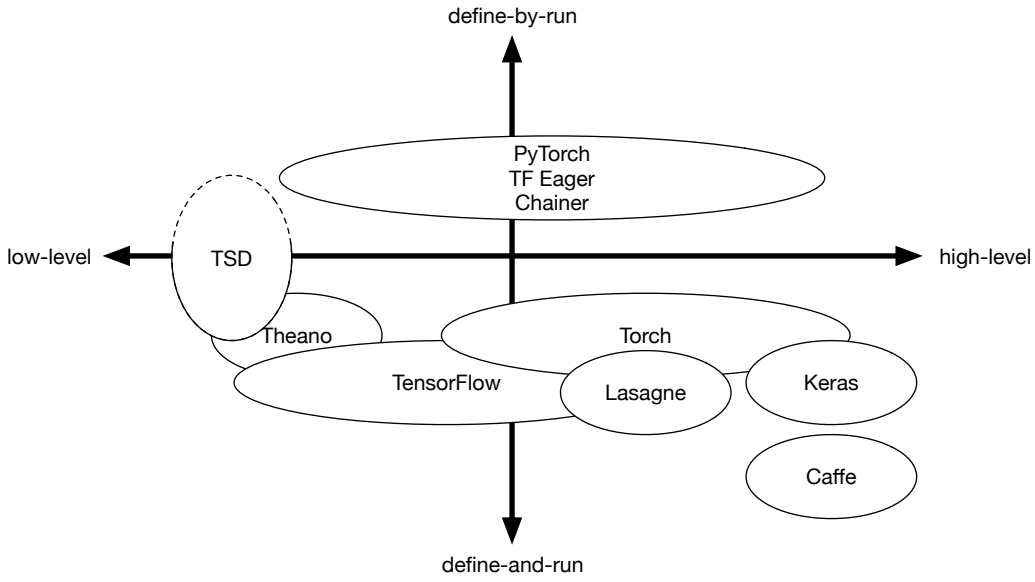


Figure 8.1: Various deep learning frameworks, revisited

As shown in Fig. 8.1, TSD occupies a peculiar position amongst other deep learning frameworks. It can be considered as 'define-and-run' but also exhibit some properties of 'define-by-run'. However, we cannot claim that TSD is also 'define-by-run' because more studies on the interaction with automatic differentiation are needed. Details will be discussed in the following paragraphs. As for the horizontal axis, TSD is sitting in an even lower level because it provides control over parallelism. Note that it is also possible to develop high-level libraries that wrap the functionality of TSD but that is for future development.

Define-and-run TSD can be considered as 'define-and-run' as one can first create a dataflow graph with cells being input placeholders and then push data into them in order to execute or train a model, as shown in section 4.1. However, there are a number of differences as well.

Firstly, graph construction in other 'define-and-run' frameworks, e.g. Theano and TensorFlow, is similar to building a syntax tree. The resulting program has a strong meta-programming feel. In contrast, the difference between a graph and a value in TSD is trans-

parent to the programmer. In TensorFlow one has to run or train the model by using an execution engine, i.e. sessions, which is not as straightforward as in TSD where one simply pushes the data and propagate the changes.

Secondly, these frameworks have been criticised for being difficult to debug because the training process works like a closed-box where intermediate values are not accessible during execution. On the contrary, TSD provides the `peek` operator, which returns the current value of any referenced nodes in a graph. This provides access to any values during execution for the purpose of debugging.

Thirdly, control flows such as conditionals and loops of the host language are not usable by these frameworks and have to be built into graphs. In TSD, this can be done by using `if` with the conditional branch being a graph. For example, the code `let x = pc 1 in if x <= 0 then t else u` creates a dataflow graph with `if` being a node inside. On the other hand, conditionals and recursion can be used in the usual sense by peeking the conditional branch. For instance, the code `let x = pc 1 in if peek x <= 0 then t else u` will only evaluate term `t`. This is because when the conditional branch is attached to a normal value, the `if`-expression will be evaluated in the normal sense.

Finally, unlike other frameworks, dataflow graphs in TSD can be thought of as dynamic since between iterations they can be modified freely by using the primitive imperative commands as shown in the example in section 4.5.

Define-by-run TSD also exhibits similar properties of 'define-by-run' as graphs are created on-the-fly during evaluation with data. *Cells* are not just input placeholders, but are also initialised with data. For example, the code `let x = ref 1 in deref x + 1` creates the graph for `x + 1` while evaluating it. Graphs in TSD are also dynamic, but they are different with those in PyTorch or Chainer. PyTorch and Chainer provides graph dynamism by building and discarding dataflow graphs every time for each iteration in the training loop. As opposed to that, one uses the imperative commands to modify the graph in TSD to

achieve dynamism. For example, during training, one could modify part of the graph as follows:

```
let x = ref 0 in
let y = ref (deref x)  in
let m = ref (deref y * pc 1 + pc 0)) in
fuse (f, ps) from m in
let grad = num_grad f ps mesloss 0.01 in
let optimise = gradient_descent ps grad 0.1 in

let training_loop = rec g. λn.
  (* changes the graph structure on some 'cond' *)
  let optimise =
    if cond
    then
      link y ...;
      fuse (f, ps) from m in
      let grad = num_grad f ps mesloss 0.01 in
      gradient_descent ps grad 0.1 in
    else
      optimise
  in
  optimise ...
```

During training, when `cond` is true, the graph structure will be modified by `link y ...`. After that, a new `optimise` function which works on the newly fused skeleton function and parameters. This means that the whole graph including the model and the gradient graphs will be reconstructed. This could be in principle more efficient than a naive implementation of 'define-by-run', for instance if the graph is static, there will be no graph reconstruction in TSD. This is also in parallel with the static subgraph optimisations provided by Chainer [23]. In Chainer, programmers are allowed to declare parts of a graph static by using the annotation `@static_code`. The program will only construct the graph structure once, but the function wrapped by `@static_code` will be called every time for each iteration. As such, 'define-by-run' first assumes dynamic construction and then introduces supports for creating static subgraphs. On the contrary, TSD is more graph-centric. Graphs in TSD are static if no changes were made to them; they are dynamic if they are modified by the imperative commands.

Idealised TensorFlow is the preliminary version of TSD. As such, many of the features, e.g. `fusion`, in TSD are based on ITF. The two languages indeed have many similarities, but they are also different in many ways.

Firstly, they both provide the `pc` operator to create model parameters. However, provisional constants in ITF are similar to global references that are always shared but not copied. This causes problems when one tries to create many parameters using recursion. For example, consider the following code:

```
let m =
  rec f. λn.
    if n <= 0 then 0 else pc 1 + f (n-1)
in
let y = m 10
```

`y` will only be connected to one provisional constant but not ten because parameters are created at compile time. As opposed to that, `pc` in TSD is an operator that creates provisional constants at runtime and can be created under recursion.

Secondly, they both provide a language construct that automatically collects model parameters. In ITF, it is called `graph abstraction` while in TSD, it is called `fusion`. One difference is that parameters are collected in compile time in ITF, meaning that it will also collect provisional constants that are actually not used by the model. For example, in the following code:

```
let f = λx. pc 1 + pc 0 in
let m = f (pc 2) in
(𝒜(f,ps). ...) m
```

`ps` contains three elements instead of just two. In contrast, parameters collection in TSD happens in runtime and only those that are connected to the model are *fused*.

Thirdly, graphs in TSD are constructed around *cells* which can be manipulated using imperative commands while that in ITF cannot be reconfigured once created. These cells allow programmers to express feedback loops in a system, which is lacking in ITF. They also

provide controls over operation and pipeline parallelism that does not exist in ITF.

Finally, their semantics are both defined based on the dynamic geometry of interaction machine. But the two machines are also different as ITF performs copying in one-go while TSD only does that when needed. Consider the following term:

```
let f = λx. x + x in
let g = λx. f x in
let h = λx. f x in
f 1
```

In ITF, both `g` and `h` gets a copy of `f` even though they are not used[1]. While in TSD, only `f 1` gets a copy. This means that TSD is theoretically more efficient than ITF.

8.4.2 Reactive programming languages

Self-adjusting computation Our work is also inspired by how changes are propagated through dataflow graphs in self-adjusting computation. However, feedback loops are not allowed in SAC because if otherwise the `stabilize()` function will diverge. In fact, circular dependencies are detected at runtime in the JaneStreet library. Instead of a one-go `stabilize()` function, TSD provides `step`-propagation, which advances one time step only. This allows for the creation of feedback loops, and the `stabilize()` function can be simulated by the following code:

$$\texttt{let stabilise} = \texttt{rec}f.\lambda x.\texttt{if step then } f\ x\ \texttt{else } 0$$

If the graph contains feedback loops such that cell values are never stabilised, this function will simply diverge.

Lustre, Lucid Synchrone and Zélus These are the languages that are the most similar to TSD in terms of graph construction and change propagation. In fact, most of the primitives in Lustre and LS can be reconstructed using TSD with option types to represent streams,

[1] Interested readers can visit `https://cwtsteven.github.io/GoI-TF-Visualiser/CBV-with-CBN-embedding/index.html` to see the step-by-step execution

i.e. `'a stream = 'a option`. The signal is absent when the current value is `None`. The comparison is shown in Table 8.1.

Lustre/LS	TSD
`x = pre t`	`let x = ref None in link x t`
`x = a -> t`	`let x = ref (Some a) in link x (root t)`
`x = a fby t`	`let x = ref (Some a) in link x t`
`x = y when c`	`let x = if c then y else None`

`x = current y`	`let z = ref y in`
	`let x = if y == None then z else y in`
	`link z x; x`

`x = merge c a b`	`let x = if c then a else b`

Table 8.1: Lustre primitives using TSD

As shown in the comparison, TSD and LS are very different in how dataflow graphs are defined. Lustre and LS are stream-oriented, where graphs are constructed through defining equations of a system. TSD however is very graph-centric, it provides imperative commands to modify the graphs at any time instant.

Chapter 9

Conclusion

In summary, we have designed a functional language for systems modelling and optimisation, *transparent synchronous dataflow*, that (1) has the ability to create straight-line code easily; (2) provides a safe way to manipulate arrays or 'tensors' under certain conditions; and (3) is built on top of a unified semantics harmonising lambda calculus and dataflow graphs. Our contributions include the design of TSD, all the theoretical results and various experimental implementations:

Language design The TSD language is

- *imperative* - dataflow graphs are explicitly manipulated using commands;
- *functional* - the language includes (call-by-value) PCF [69] as a subset;
- *simply-typed* - the type system is essentially that of PCF;
- *modal* - dataflow graphs are constructed and can be executed separately;
- *eager* - dataflow graphs are constructed on-the-fly by computation at runtime;
- *parallel* - dataflow graphs can be executed in parallel;
- *dynamic* - dataflow graphs can be changed during execution;
- *transparent* - the distinction between a graph and a value is transparent to programmers;
- *synchronous* - the change propagation is safe with feedback loops; and
- *symmetric* - the result of a program is invariant to permutations of parameters under

certain conditions.

Theoretical results Our theoretical contributions are:

- *specification* - we give a precise mathematical definition of the language using an *abstract machine*, harmonising the semantics of the PCF-like language with that of dataflow;
- *determinism* - we show that the parallel evaluation of the dataflow graph produces deterministic results;
- *safety* - we show that for any well-typed program, the abstract machine executes without reaching illegal states;
- *termination* - we show that the execution of any recursion-free program with dataflow terminates by reaching the final state of the abstract machine;
- *symmetry* - we show that the result of any program (under certain conditions) is independent of how parameters are collected;
- *equivalences* - we show that beta-equivalence for usual call-by-value languages hold in our language; and
- *efficiency* - we show that no operation is 'expensive' in terms of time and memory utilisation.

Methodological results We have also built the following experimental implementations to show that the language is expressive and efficient:

- a native compiler for DGoIM implementing lambda calculus with natural numbers; and
- an OCaml library together with PPX extensions for the TSD language.

9.1 Future Work

In this final section, we will discuss several possible future extensions to this thesis that are not introduced in previous chapters.

Tensors Tensors can be seen as generalised scalars and vectors that are frequently used in machine learning framework to model a dataset. For example, a set of 2D-images can be represented by a rank-3 tensor. It is possible to generalise the TSD language such that it also supports tensors and tensor operations. For every specific rank and shape the tensor space and element-wise tensor operations form a field which is the core primitive data in TSD. The extension would also require bringing tensor product into the language. An alternative is to extend the TSD language with arrays or lists in order to implement tensors and tensor operations in terms of them.

Automatic differentiation One of the main objectives of this thesis is to design a language that provides an easy way to create straight-line code for the purpose of automatic differentiation. Recall that in the definition of gradient descent `grad_desc` function shown in section 4.1, it takes a function `grad` which creates the gradient graph at runtime. It is possible to add another primitive operation `autodiff` which automatically creates the graph that calculates the partial derivatives. For models that do not have intermediate cells, automatic differentiation is simple as the graph represent a straight-line code which consist of differentiable operators. Conventional AD techniques such as back propagation could be easily applied. For models that contains pipeline (cells) their input values and intermediate values will have to be kept by creating a pipeline in the gradient graph since the calculation of the derivatives would require this information. For models that contain feedback loops, creating the gradient graph is more subtle because the derivatives are dependent on previous states. More studies need to be done to see how common differentiation techniques for recurrent neural network such as back propagation through time could work in the TSD language.

Distributed computing and automatic parallelisation As mentioned in section 4.1 the TSD language provides fine-grained controls over operation and pipeline parallelism through *cells*. It is also possible to distribute these cells to different processors or machines in order to

execute them simultaneously, as it is guaranteed by the semantics that they can be executed in parallel without conflicts. A possible program is shown as follows:

```
let x = ref@A 1 in
let y = ref@B (deref x + 1) in
⋮
```

where `ref@A` sends the cell to location `A`. This is possible as distributed computing via geometry of interaction has already been created in [36]. It is also possible to let the runtime decide how and where to distribute the cells based on the graph structure or some profile information about the model.

Visual debugging One of the advantages of using a graph abstract machine to design languages is that the execution of a program can be easily visualised. As an example, a visualiser for the dataflow core of the TSD language[1] has been created. It is possible to enhance the visualiser with functionalities such as adding break points and reversing the execution. In fact, a prototype visual debugger for a PCF-like language has already been developed[2].

Graph optimisation and contextual equivalences Up till now we have only proved that beta-equivalence hold in our language, but it is possible to prove other theorems as well since we have the formalism for proving program equivalences, for example garbage collection.

Native interpreter In chapter 7 we have outlined the implementation of a native compiler that compiles the source code directly into self-modifying x86 assembly codes. However, its performance is unsatisfactory, as self-modifying codes are generally problematic and slow under current computer architecture. On the other hand, we have implemented the TSD language as an OCaml library together with PPX extension. However, this implementation does not follow the specification of the graph machine faithfully, but instead opted for better performance. The major difference is that fusion of parameters is done at compile time

[1] `https://github.com/cwtsteven/TSD-visual`
[2] `https://fyp.jackhughesweb.com/`

instead of runtime, which makes sense when the model is huge and traversing the graph at runtime might hurt the performance. Also, because the OCaml implementation is an embedded DSL within OCaml, the construction of the dataflow graph is not as seamless as the original TSD language since programmers need to *lift* constants and functions manually so that they can be used to create the graphs.

It is therefore in our interest to create an interpreter that faithfully implements the TSD language and is also efficient. One possible way is to implement the abstract machine as the runtime system and compile source code into its graph representation.

Bibliography

[1] Martín Abadi et al. "TensorFlow: A System for Large-Scale Machine Learning". In: *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*. Ed. by Kimberly Keeton and Timothy Roscoe. USENIX Association, 2016, pp. 265–283. URL: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/abadi.

[2] Umut A. Acar, Matthias Blume, and Jacob Donham. "A consistent semantics of self-adjusting computation". In: *J. Funct. Program.* 23.3 (2013), pp. 249–292. DOI: 10.1017/S0956796813000099. URL: https://doi.org/10.1017/S0956796813000099.

[3] Beniamino Accattoli. "The Complexity of Abstract Machines". In: *Proceedings Third International Workshop on Rewriting Techniques for Program Transformations and Evaluation, WPTE@FSCD 2016, Porto, Portugal, 23rd June 2016*. 2016, pp. 1–15. DOI: 10.4204/EPTCS.235.1. URL: https://doi.org/10.4204/EPTCS.235.1.

[4] Akshay Agrawal et al. "TensorFlow Eager: A multi-stage, Python-embedded DSL for machine learning". In: *Proceedings of Machine Learning and Systems 2019, MLSys 2019, Stanford, CA, USA, March 31 - April 2, 2019*. Ed. by Ameet Talwalkar, Virginia Smith, and Matei Zaharia. mlsys.org, 2019. URL: https://proceedings.mlsys.org/book/286.pdf.

[5] Johan Åkesson. "Optimica—an extension of modelica supporting dynamic optimization". In: *Proc. 6th International Modelica Conference 2008*. 2008.

[6] Rami Al-Rfou et al. "Theano: A Python framework for fast computation of mathematical expressions". In: *CoRR* abs/1605.02688 (2016). arXiv: 1605.02688. URL: http://arxiv.org/abs/1605.02688.

[7] Edward A. Ashcroft and William W. Wadge. "Lucid, a Nonprocedural Language with Iteration". In: *Commun. ACM* 20.7 (1977), pp. 519–526. DOI: 10.1145/359636.359715. URL: https://doi.org/10.1145/359636.359715.

[8] Muhammad Awais. *RNN LSTM and Deep Learning Libraries*. URL: https://udrc.eng.ed.ac.uk/sites/udrc.eng.ed.ac.uk/files/attachments/UDRC_RNN_LSTM_LibrariesTutorial.pdf.

[9] Adrian Baez-Ortega. Feb. 2019. URL: https://baezortega.github.io/2019/02/05/kepler-orbits/.

[10] Engineer Bainomugisha et al. "A survey on reactive programming". In: *ACM Comput. Surv.* 45.4 (2013), 52:1–52:34. DOI: 10.1145/2501654.2501666. URL: https://doi.org/10.1145/2501654.2501666.

[11] Atilim Gunes Baydin et al. "Automatic Differentiation in Machine Learning: a Survey". In: *J. Mach. Learn. Res.* 18 (2017), 153:1–153:43. URL: http://jmlr.org/papers/v18/17-468.html.

[12] Albert Benveniste et al. "The synchronous languages 12 years later". In: *Proceedings of the IEEE* 91.1 (2003), pp. 64–83.

[13] Gérard Berry and Laurent Cosserat. "The ESTEREL Synchronous Programming Language and its Mathematical Semantics". In: *Seminar on Concurrency, Carnegie-Mellon University, Pittsburg, PA, USA, July 9-11, 1984*. Ed. by Stephen D. Brookes, A. W. Roscoe, and Glynn Winskel. Vol. 197. Lecture Notes in Computer Science. Springer, 1984, pp. 389–448. DOI: 10.1007/3-540-15670-4_19. URL: https://doi.org/10.1007/3-540-15670-4%5C_19.

[14] Rick Bitter, Taqi Mohiuddin, and Matt Nawrocki. *LabVIEW: Advanced programming techniques*. Crc Press, 2006.

[15] Adnan Bouakaz, Pascal Fradet, and Alain Girault. "A Survey of Parametric Dataflow Models of Computation". In: *ACM Trans. Design Autom. Electr. Syst.* 22.2 (2017), 38:1–38:25. DOI: 10.1145/2999539. URL: https://doi.org/10.1145/2999539.

[16] Timothy Bourke and Marc Pouzet. "Zélus: a synchronous language with ODEs". In: *Proceedings of the 16th international conference on Hybrid systems: computation and control, HSCC 2013, April 8-11, 2013, Philadelphia, PA, USA*. Ed. by Calin Belta and Franjo Ivancic. ACM, 2013, pp. 113–118. DOI: 10.1145/2461328.2461348. URL: https://doi.org/10.1145/2461328.2461348.

[17] Jason Brownlee. *A Gentle Introduction to Backpropagation Through Time*. June 2017. URL: https://machinelearningmastery.com/gentle-introduction-backpropagation-time/.

[18] Caffe. *LeNet MNIST Tutorial*. URL: https://github.com/BVLC/caffe/tree/master/examples/mnist.

[19] Glennan Carnie. *An introduction to C++ templates*. URL: https://blog.feabhas.com/2014/05/an-introduction-to-c-templates/.

[20] Paul Caspi and Marc Pouzet. "A Co-iterative Characterization of Synchronous Stream Functions". In: *First Workshop on Coalgebraic Methods in Computer Science, CMCS 1998, Lisbon, Portugal, March 28-29, 1998*. Ed. by Bart Jacobs et al. Vol. 11. Electronic Notes in Theoretical Computer Science. Elsevier, 1998, pp. 1–21. DOI: 10.1016/S1571-0661(04)00050-7. URL: https://doi.org/10.1016/S1571-0661(04)00050-7.

[21] Paul Caspi and Marc Pouzet. "A functional extension of Lustre". In: 1995.

[22] Paul Caspi et al. "Lustre: A Declarative Language for Programming Synchronous Systems". In: *Conference Record of the Fourteenth Annual ACM Symposium on Principles of Programming Languages, Munich, Germany, January 21-23, 1987*. ACM Press, 1987, pp. 178–188. DOI: 10.1145/41625.41641. URL: https://doi.org/10.1145/41625.41641.

[23] Chainer. *Static Subgraph Optimizations: Design Notes*. URL: https://docs.chainer.org/en/stable/reference/static_graph_design.html.

[24] Steven Cheung et al. "A Functional Perspective on Machine Learning via Programmable Induction and Abduction". In: *Functional and Logic Programming - 14th International Symposium, FLOPS 2018, Nagoya, Japan, May 9-11, 2018, Proceedings*. Ed. by John P. Gallagher and Martin Sulzmann. Vol. 10818. Lecture Notes in Computer Science. Springer, 2018, pp. 84–98. DOI: 10.1007/978-3-319-90686-7_6. URL: https://doi.org/10.1007/978-3-319-90686-7%5C_6.

[25] Derek Chia. *A line-by-line layman's guide to Linear Regression using TensorFlow*. Medium. 2018. URL: https://towardsdatascience.com/a-line-by-line-laymans-guide-to-linear-regression-using-tensorflow-3c0392aa9e1f.

[26] François Chollet et al. *Keras*. https://keras.io. 2015.

[27] Jean-Louis Colaço, Bruno Pagano, and Marc Pouzet. "A conservative extension of synchronous data-flow with state machines". In: *EMSOFT 2005, September 18-22, 2005, Jersey City, NJ, USA, 5th ACM International Conference On Embedded Software, Proceedings*. Ed. by Wayne H. Wolf. ACM, 2005, pp. 173–182. DOI: 10.1145/1086228.1086261. URL: https://doi.org/10.1145/1086228.1086261.

[28] R. Collobert, K. Kavukcuoglu, and C. Farabet. "Torch7: A Matlab-like Environment for Machine Learning". In: *BigLearn, NIPS Workshop*. 2011.

[29] Gregory H. Cooper and Shriram Krishnamurthi. "Embedding Dynamic Dataflow in a Call-by-Value Language". In: *Programming Languages and Systems, 15th European Symposium on Programming, ESOP 2006, Held as Part of the Joint European Conferences on Theory and Practice of Software,*

ETAPS 2006, Vienna, Austria, March 27-28, 2006, Proceedings. Ed. by Peter Sestoft. Vol. 3924. Lecture Notes in Computer Science. Springer, 2006, pp. 294–308. DOI: 10.1007/11693024_20. URL: https://doi.org/10.1007/11693024%5C_20.

[30] Luís Damas and Robin Milner. "Principal Type-Schemes for Functional Programs". In: *Conference Record of the Ninth Annual ACM Symposium on Principles of Programming Languages, Albuquerque, New Mexico, USA, January 1982*. Ed. by Richard A. DeMillo. ACM Press, 1982, pp. 207–212. DOI: 10.1145/582153.582176. URL: https://doi.org/10.1145/582153.582176.

[31] Vincent Danos and Laurent Regnier. "Reversible, Irreversible and Optimal Lambda-machines". In: *Electron. Notes Theor. Comput. Sci.* 3 (1996), pp. 40–60. DOI: 10.1016/S1571-0661(05)80402-5. URL: https://doi.org/10.1016/S1571-0661(05)80402-5.

[32] Sander Dieleman et al. *Lasagne: First release.* Aug. 2015. DOI: 10.5281/zenodo.27878. URL: http://dx.doi.org/10.5281/zenodo.27878.

[33] Simulink Documentation. *Simulation and Model-Based Design*. 2020. URL: https://www.mathworks.com/products/simulink.html.

[34] Derek Dreyer. "A type system for well-founded recursion". In: *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2004, Venice, Italy, January 14-16, 2004*. Ed. by Neil D. Jones and Xavier Leroy. ACM, 2004, pp. 293–305. DOI: 10.1145/964001.964026. URL: https://doi.org/10.1145/964001.964026.

[35] Conal Elliott and Paul Hudak. "Functional Reactive Animation". In: *Proceedings of the 1997 ACM SIGPLAN International Conference on Functional Programming (ICFP '97), Amsterdam, The Netherlands, June 9-11, 1997*. Ed. by Simon L. Peyton Jones, Mads Tofte, and A. Michael Berman. ACM, 1997, pp. 263–273. DOI: 10.1145/258948.258973. URL: https://doi.org/10.1145/258948.258973.

[36] Olle Fredriksson and Dan R. Ghica. "Seamless Distributed Computing from the Geometry of Interaction". In: *Trustworthy Global Computing - 7th International Symposium, TGC 2012, Newcastle upon Tyne, UK, September 7-8, 2012, Revised Selected Papers*. Ed. by Catuscia Palamidessi and Mark Dermot Ryan. Vol. 8191. Lecture Notes in Computer Science. Springer, 2012, pp. 34–48. DOI: 10.1007/978-3-642-41157-1_3. URL: https://doi.org/10.1007/978-3-642-41157-1%5C_3.

[37] Peter A. Fritzson. *Principles of object-oriented modeling and simulation with Modelica 2.1*. Wiley, 2004. ISBN: 978-0-471-47163-9.

[38] Dan R. Ghica. "Geometry of synthesis: a structured approach to VLSI design". In: *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2007, Nice, France, January 17-19, 2007*. Ed. by Martin Hofmann and Matthias Felleisen. ACM, 2007, pp. 363–375. DOI: 10.1145/1190216.1190269. URL: https://doi.org/10.1145/1190216.1190269.

[39] Dan R. Ghica, Koko Muroya, and Todd Waugh Ambridge. "Local Reasoning for Robust Observational Equivalence". In: *CoRR* abs/1907.01257 (2019). arXiv: 1907.01257. URL: http://arxiv.org/abs/1907.01257.

[40] Dan R. Ghica, Alex I. Smith, and Satnam Singh. "Geometry of synthesis iv: compiling affine recursion into static hardware". In: *Proceeding of the 16th ACM SIGPLAN international conference on Functional Programming, ICFP 2011, Tokyo, Japan, September 19-21, 2011*. Ed. by Manuel M. T. Chakravarty, Zhenjiang Hu, and Olivier Danvy. ACM, 2011, pp. 221–233. DOI: 10.1145/2034773.2034805. URL: https://doi.org/10.1145/2034773.2034805.

[41] Jean-Yves Girard. "Geometry of Interaction 1: Interpretation of System F". In: *Logic Colloquium '88*. Ed. by R. Ferro et al. Vol. 127. Studies in Logic and the Foundations of Mathematics. Elsevier, 1989, pp. 221–260. DOI: https://doi.org/10.1016/S0049-237X(08)70271-4. URL: http://www.sciencedirect.com/science/article/pii/S0049237X08702714.

[42] Jean-Yves Girard. "Linear Logic". In: *Theor. Comput. Sci.* 50 (1987), pp. 1–102. DOI: 10.1016/0304-3975(87)90045-4. URL: https://doi.org/10.1016/0304-3975(87)90045-4.

[43] Andreas Griewank et al. "On automatic differentiation". In: *Mathematical Programming: recent developments and applications* 6.6 (1989), pp. 83–107.

[44] Bucker H., K. Buschelman, and P. Hovl. "A Matrix-Matrix Multiplication Approach to the Automatic Differentiation and Parallelization of Straight-Line Codes". In: (Nov. 2000).

[45] N. Halbwachs et al. "The synchronous data flow programming language LUSTRE". In: *Proceedings of the IEEE* 79.9 (1991), pp. 1305–1320.

[46] William G. Hatcher and Wei Yu. "A Survey of Deep Learning: Platforms, Applications and Emerging Research Trends". In: *IEEE Access* 6 (2018), pp. 24411–24432. DOI: 10.1109/ACCESS.2018.2830661. URL: https://doi.org/10.1109/ACCESS.2018.2830661.

[47] J. Roger Hindley. "The principal type-scheme of an object in combinatory logic". In: *Transactions of the American Mathematical Society* 146 (1969), pp. 29–66.

[48] Naohiko Hoshino, Koko Muroya, and Ichiro Hasuo. "Memoryful geometry of interaction: from coalgebraic components to algebraic effects". In: *Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS), CSL-LICS '14, Vienna, Austria, July 14 - 18, 2014*. Ed. by Thomas A. Henzinger and Dale Miller. ACM, 2014, 52:1–52:10. DOI: 10.1145/2603088.2603124. URL: https://doi.org/10.1145/2603088.2603124.

[49] https://www.amd.com/system/files/TechDocs/24592.pdf. *AMD64 Architecture Programmer's Manual Volume 1:Application Programming*. Tech. rep. Advanced Micro Devices Inc., Dec. 2017. URL: https://www.amd.com/system/files/TechDocs/24592.pdf.

[50] Janestreet. *janestreet/incremental*. June 2015. URL: https://github.com/janestreet/incremental.

[51] Yangqing Jia et al. "Caffe: Convolutional Architecture for Fast Feature Embedding". In: *Proceedings of the ACM International Conference on Multimedia, MM '14, Orlando, FL, USA, November 03 - 07, 2014*. Ed. by Kien A. Hua et al. ACM, 2014, pp. 675–678. DOI: 10.1145/2647868.2654889. URL: https://doi.org/10.1145/2647868.2654889.

[52] Wesley M. Johnston, J. R. Paul Hanna, and Richard J. Millar. "Advances in dataflow programming languages". In: *ACM Comput. Surv.* 36.1 (2004), pp. 1–34. DOI: 10.1145/1013208.1013209. URL: https://doi.org/10.1145/1013208.1013209.

[53] Harshit Kumar. Feb. 2019. URL: https://kharshit.github.io/blog/2019/02/22/backpropagation-through-time.

[54] Ugo Dal Lago and Ulrich Schöpp. "Computation by interaction for space-bounded functional programming". In: *Inf. Comput.* 248 (2016), pp. 150–194. DOI: 10.1016/j.ic.2015.04.006. URL: https://doi.org/10.1016/j.ic.2015.04.006.

[55] Zachary Chase Lipton. "A Critical Review of Recurrent Neural Networks for Sequence Learning". In: *CoRR* abs/1506.00019 (2015). arXiv: 1506.00019. URL: http://arxiv.org/abs/1506.00019.

[56] Moshe Looks et al. "Deep Learning with Dynamic Computation Graphs". In: *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017. URL: https://openreview.net/forum?id=ryrGawqex.

[57] Ian Mackie. "The Geometry of Interaction Machine". In: *Conference Record of POPL'95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, California,*

USA, January 23-25, 1995. Ed. by Ron K. Cytron and Peter Lee. ACM Press, 1995, pp. 198–208. DOI: 10.1145/199448.199483. URL: https://doi.org/10.1145/199448.199483.

[58] Louis Mandel and Marc Pouzet. "ReactiveML: a reactive extension to ML". In: *Proceedings of the 7th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, July 11-13 2005, Lisbon, Portugal*. Ed. by Pedro Barahona and Amy P. Felty. ACM, 2005, pp. 82–93. DOI: 10.1145/1069774.1069782. URL: https://doi.org/10.1145/1069774.1069782.

[59] Bart van Merriënboer et al. "Blocks and Fuel: Frameworks for deep learning". In: *CoRR* abs/1506.00619 (2015). arXiv: 1506.00619. URL: http://arxiv.org/abs/1506.00619.

[60] Leo A. Meyerovich et al. "Flapjax: a programming language for Ajax applications". In: *Proceedings of the 24th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2009, October 25-29, 2009, Orlando, Florida, USA*. Ed. by Shail Arora and Gary T. Leavens. ACM, 2009, pp. 1–20. DOI: 10.1145/1640089.1640091. URL: https://doi.org/10.1145/1640089.1640091.

[61] Robin Milner. "A Theory of Type Polymorphism in Programming". In: *J. Comput. Syst. Sci.* 17.3 (1978), pp. 348–375. DOI: 10.1016/0022-0000(78)90014-4. URL: https://doi.org/10.1016/0022-0000(78)90014-4.

[62] Koko Muroya, Steven W. T. Cheung, and Dan R. Ghica. "The Geometry of Computation-Graph Abstraction". In: *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018*. Ed. by Anuj Dawar and Erich Grädel. ACM, 2018, pp. 749–758. DOI: 10.1145/3209108.3209127. URL: https://doi.org/10.1145/3209108.3209127.

[63] Koko Muroya and Dan R. Ghica. "The Dynamic Geometry of Interaction Machine: A Call-by-Need Graph Rewriter". In: *26th EACSL Annual Conference on Computer Science Logic, CSL 2017, August 20-24, 2017, Stockholm, Sweden*. Ed. by Valentin Goranko and Mads Dam. Vol. 82. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017, 32:1–32:15. DOI: 10.4230/LIPIcs.CSL.2017.32. URL: https://doi.org/10.4230/LIPIcs.CSL.2017.32.

[64] Koko Muroya and Dan R. Ghica. "The Dynamic Geometry of Interaction Machine: A Token-Guided Graph Rewriter". In: *Logical Methods in Computer Science* 15.4 (2019). URL: https://lmcs.episciences.org/5882.

[65] Giang Nguyen et al. “Machine Learning and Deep Learning frameworks and libraries for large-scale data mining: a survey”. In: *Artif. Intell. Rev.* 52.1 (2019), pp. 77–124. DOI: 10.1007/s10462-018-09679-z. URL: https://doi.org/10.1007/s10462-018-09679-z.

[66] Johan Nordlander, Magnus Carlsson, and Andy Gill. “Unrestricted pure call-by-value recursion”. In: *Proceedings of the ACM Workshop on ML, 2008, Victoria, BC, Canada, September 21, 2008*. Ed. by Eijiro Sumii. ACM, 2008, pp. 23–34. DOI: 10.1145/1411304.1411309. URL: https://doi.org/10.1145/1411304.1411309.

[67] Joachim Parrow and Björn Victor. “The Fusion Calculus: Expressiveness and Symmetry in Mobile Processes”. In: *Thirteenth Annual IEEE Symposium on Logic in Computer Science, Indianapolis, Indiana, USA, June 21-24, 1998*. IEEE Computer Society, 1998, pp. 176–185. DOI: 10.1109/LICS.1998.705654. URL: https://doi.org/10.1109/LICS.1998.705654.

[68] Adam Paszke et al. “Automatic differentiation in PyTorch”. In: (2017).

[69] Gordon D. Plotkin. “LCF Considered as a Programming Language”. In: *Theor. Comput. Sci.* 5.3 (1977), pp. 223–255. DOI: 10.1016/0304-3975(77)90044-5. URL: https://doi.org/10.1016/0304-3975(77)90044-5.

[70] G. Ramalingam and Thomas W. Reps. “A Categorized Bibliography on Incremental Computation”. In: *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Charleston, South Carolina, USA, January 1993*. Ed. by Mary S. Van Deusen and Bernard Lang. ACM Press, 1993, pp. 502–510. DOI: 10.1145/158511.158710. URL: https://doi.org/10.1145/158511.158710.

[71] Roshan Santhosh. *Linear Regression In Theano*. URL: https://roshansanthosh.wordpress.com/2015/02/22/linear-regression-in-theano/.

[72] Ulrich Schöpp. “Space-Efficient Computation by Interaction”. In: *Computer Science Logic, 20th International Workshop, CSL 2006, 15th Annual Conference of the EACSL, Szeged, Hungary, September 25-29, 2006, Proceedings*. Ed. by Zoltán Ésik. Vol. 4207. Lecture Notes in Computer Science. Springer, 2006, pp. 606–621. DOI: 10.1007/11874683_40. URL: https://doi.org/10.1007/11874683%5C_40.

[73] Richard Socher et al. “Parsing Natural Scenes and Natural Language with Recursive Neural Networks”. In: *Proceedings of the 28th International Conference on Machine Learning, ICML 2011, Bellevue, Washington, USA, June 28 - July 2, 2011*. Ed. by Lise Getoor and Tobias Scheffer. Omnipress, 2011, pp. 129–136. URL: https://icml.cc/2011/papers/125%5C_icmlpaper.pdf.

[74] S. Sumaryo, A. Halim, and K. Ramli. "Improved discrete event simulation model of traffic light control on a single intersection". In: *2013 International Conference on QiR*. 2013, pp. 116–120. DOI: 10.1109/QiR.2013.6632548.

[75] TensorFlow. *Introduction to Gradients and Automatic Differentiation*. URL: https://www.tensorflow.org/guide/autodiff.

[76] Seiya Tokui et al. "Chainer: A Deep Learning Framework for Accelerating the Research Cycle". In: *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, KDD 2019, Anchorage, AK, USA, August 4-8, 2019*. Ed. by Ankur Teredesai et al. ACM, 2019, pp. 2002–2011. DOI: 10.1145/3292500.3330756. URL: https://doi.org/10.1145/3292500.3330756.

[77] Torch. *Training a neural network*. URL: https://github.com/torch/nn/blob/master/doc/training.md#nn.traningneuralnet.dok.

[78] Hao Zhang. July 2018. URL: https://petuum.medium.com/intro-to-dynamic-neural-networks-and-dynet-67694b18cb23.