Fast Graph Programs

Project Report

Graham Campbell, Jack Romo, and Detlef Plump

Department of Computer Science
University of York, United Kingdom

Summer 2018
Contents

Prologue 3

1 Introduction 4
   1.1 Background .................................................. 4
   1.2 Project Aims .................................................. 5

2 Automata Simulation 6
   2.1 Motivation .................................................... 6
   2.2 FSAs .......................................................... 6
   2.3 DPDAs .......................................................... 10

3 Efficient Topological Sort 13
   3.1 Problem Description .......................................... 13
   3.2 GP2 Implementation ........................................... 14
   3.3 Complexity Results .......................................... 16

4 Efficient Red-Black Trees 18
   4.1 Problem Description .......................................... 18
   4.2 Functional RBTs .............................................. 18
   4.3 GP2 Implementation .......................................... 19
   4.4 Complexity Results .......................................... 28
5 Program Refinement

5.1 Motivation ................................................................. 29
5.2 Refinement I ............................................................... 31
5.3 Refinement II .............................................................. 33
5.4 Refinement III ............................................................ 34
5.5 Refinement of List Sorting ............................................. 35

6 Conclusion ................................................................. 38

A Project Tooling ............................................................. 39

A.1 Overview ................................................................. 39
A.2 GP2I ................................................................. 40
A.3 GP2 Bench ............................................................. 42
A.4 LaTeX ................................................................. 44

B Proofs ................................................................. 47

B.1 Complexity of RBT Insertion ......................................... 47

Bibliography ................................................................. 48
This report summarizes the technical results of an undergraduate project carried out over 10 weeks by Graham Campbell and Jack Romo. The project was supervised by Dr. Detlef Plump and funded by a Vacation Bursary of the Engineering and Physical Sciences Research Council (EPSRC).

The project’s aim was to demonstrate with case studies that rule-based graph programs in the language GP2 can be made fast by using so-called root nodes. The basic idea is that by equipping both rules and host graphs with roots that correspond to each other, the non-determinism in rule matching can be greatly reduced.

As a “warm-up”, the first case study shows that rooted graph programs can simulate finite automata and deterministic pushdown automata in a natural way. The transition diagrams of such automata are directed graphs anyway and program execution proceeds by moving the root, which represents the current state, along the transitions in the diagram.

In the second case study, topological sorting on acyclic graphs is implemented by a rooted GP2 program that encodes a depth-first traversal strategy. Benchmarking on input graphs of up to 200,000 nodes shows that this program runs in linear time on acyclic graphs of bounded node degree (under some mild assumptions such as connectedness and a unique node of indegree 0). This is a novel result in the field of rule-based graph transformation.

The third case study represents red-black trees as GP2 graphs and presents a rooted graph program that generates a red-black tree for $n$ integers in time $O(n \log n)$. Again, this result is novel and has been empirically confirmed on input graphs of up to 100,000 nodes.

Finally, this report describes some initial attempts to refine unrooted graph programs to faster rooted programs. For this, the problem of reversing all edges in an input graph is considered. Rooted graph programs are presented that accomplish edge reversal on lists, trees and forests in linear time. In contrast, the corresponding unrooted programs have a quadratic running time.
Chapter 1

Introduction

1.1 Background

Graphs are used to model dynamic structures in areas of computer science such as
compiler construction, pointer programming, natural language processing, and model-
driven software development. The behaviour of systems in such areas can often be
captured by graph transformation rules specifying small state changes which are typically
of constant size. Rule-based languages for problem solving by graph transformation
include AGG [RET12], GReAT [AKN+06], GROOVE [GdMR+12], GrGen.Net [JBK10],
Henshin [ABJ+10] and PORGY [FKMP14]. This report presents algorithms in the graph
programming language GP2 [Plu12, Bak15], which has been designed to support formal
reasoning on programs.

The declarative nature of graph transformation rules comes at the price of high execution
times caused by graph matching. In general, to match the left-hand graph $L$ of a
rule within a host graph $G$ requires time $\text{size}(G)\text{size}(L)$. As a consequence, linear graph
algorithms are usually slowed down to polynomial complexity when recast as programmed
graph transformation systems. To speed up graph matching, Dörr [Dör95] proposed to
equip rules and host graphs with distinguished nodes, so-called roots, and to match roots
in rules with roots in host graphs. This concept has been implemented in GP2 and in
a case study shown to allow a 2-colouring program that runs in linear time on graphs of
bounded node degree [BP16].
1.2 Project Aims

The main aim of this project is to explore the application of root nodes to a number of well-known graph algorithms. The goal is to demonstrate how root nodes may achieve standard running times, reducing the overhead of subgraph matching to a constant factor. By exploring several case studies, we hope to show what properties one should expect from an algorithm that can be speeded up by root nodes. We also hope to derive some design patterns for programs using root nodes, and establish how intuitive their usage is.

Our secondary goal is to explore the refinement of unrooted GP2 programs by introducing root nodes. Clearly, the use of root nodes as a programming construct lessens the elegance of unrooted programs, exposing internal implementation beyond the theoretical semantics. Ideally, one would like to ignore root nodes as a construct altogether, let the user writing programs with no awareness of anything beyond semantics, and the GP2 compiler augmenting programs to use root nodes for faster performance. It is our hope that some simple programs may possess a clear method for their refinement to a rooted version, shedding light on this issue and incentivising future work on more sophisticated refinement techniques, along with their potential mechanisation.
Chapter 2

Automata Simulation

2.1 Motivation

Our exploration of graph programs using root nodes begins at the well-known problem of automata simulation. It should be intuitive that automata problems translate well to graph programming, due to their elegant representation in graphical form.

Simulating execution of automata is of special interest to root nodes due to its locality; we need only consider what state we are currently in, the next character to read in our string, and the next states we may progress to. We demonstrate here how these properties can be exploited with root nodes in an intuitive way, cementing a strong case for the effectiveness of root nodes in graph programming.

2.2 FSAs

A finite-state automaton, or FSA, is defined as a 5-tuple of the form

\[ M = (Q, \Sigma, q_0, A, \delta) \]  

(2.1)

where \( Q \) is the finite set of states, \( \Sigma \) is the finite input alphabet, \( q_0 \in S \) is the initial state, \( \delta : Q \times \Sigma \to Q \) is the transition function and \( A \subseteq Q \) is the set of accepting states. [Mar03] One can think of an FSA moving from the state \( q_1 \in Q \) to the state \( q_2 \in Q \) by reading the character \( \sigma \in \Sigma \) when \( \delta(q_1, \sigma) = q_2 \). We are interested in whether, starting at the initial state \( q_0 \) and an input string \( y \in \Sigma^* \), sequentially reading each character in \( y \) will result in us moving between states such that we stop at one in \( A \); If this is the case, we say the automata accepts the string \( y \).

The crucial property of automata for us is their simple representation as a graph. We may represent each state in \( Q \) with a vertex in the graph, and a directed edge labelled with \( \sigma \in \Sigma \) to be drawn between the vertices of two states \( q_1, q_2 \in Q \) if \( \delta(q_1, \sigma) = q_2 \).
This allows us to understand execution of an automata as a graph traversal problem: starting from the initial state's vertex, we move along edges that correspond to each character in the input string until no further progress can be made, then check which state we end up in. Such an algorithm can be more concretely understood as the following:

\[
\begin{align*}
S & := \text{input string} \\
G & := \text{graph of FSA} \\
Q & := \text{initial state of FSA}
\end{align*}
\]

while $S$ is non-empty:

\[
\begin{align*}
E & := \{\text{edges outgoing from } Q \text{ in } G\} \\
\text{if any } e \text{ in } E \text{ such that } \text{label}(e) = \text{head}(S): \\
Q & := \text{target vertex of } e \\
S & := \text{tail}(S) \\
\text{else:} \\
\text{fail}
\end{align*}
\]

return $Q$

Figure 2.1: Pesudo Code FSA Execution

The reader should note we ignore detection of final states - checking if one is reached is a trivial task once the last state reached is known.

When executing an FSA as outlined above, at any given point in time, we only need to be immediately aware of three things:

- The current state ($Q$);
- The next character to be read ($\text{head}(S)$);
- The set of outgoing edges from our state's vertex and their target states ($E$).

Since $\delta$ is a function, we can only have up to $|\Sigma|$ edges outgoing from any vertex in our FSA graph. This imposes a constant upper bound on how many edges leave any vertex in an FSA graph, due to $\Sigma$ being finite. The reader may now note how this corresponds to the restriction we have placed on rooted graph programs earlier, in that we desire our host graphs to have a bounded degree.

It is at this point we may begin to piece together how a GP2 program simulating an FSA's execution on an input string might work. Our host graph will consist of two components:

- The FSA graph, where each state corresponds to one unique vertex with the state as its label, and an edge labelled with $\sigma$ between two vertices $q_1, q_2$ if $\delta(q_1, \sigma) = q_2$,
- The input string, written as a linked list where each vertex is labelled in sequence with each character in the input string.
Note that exactly what we mentioned needs to be immediately accessible at all times now directly corresponds to local areas of this host graph; more specifically, the first vertex in the input string list, the vertex of the state we are in and its immediate neighbourhood must be accessed at every step of computation. We can immediately root the head of our string and the current state’s vertex in our graph to take advantage of this. At any one step, we advance both root nodes to the next character and the next state we move to. Note that this means the host graph should have the initial state $q_0$ rooted to begin with. We will also choose to distinguish our input string from our FSA graph by colouring it entirely blue, and will append an extra vertex labelled with ’X’ on the end of it for reasons discussed later.

Below is an example host graph for an FSA taking the input string "ab", to intuitively solidify our constructions:

![Figure 2.2: An Example FSA Host Graph](image)

We are now ready to present our GP2 program to simulate FSAs. The program is as follows:

```
Main = {step, stepL}!
```

![Figure 2.3: GP2 FSA Program](image)

This program is especially simple, seeing as all we wish to do is advance to new states continuously. We merely perform whatever advancements are possible as long as we can, represented by the two rules `step` and `stepL`; these each represent advancing to a new state or following a loop edge back to the same state, respectively. The requirement that matches are injective means we must distinguish these cases. Note the use of comma-separated statements in curly brackets to represent angelic nondeterminism.
Most of the complexity is contained within the rules \textit{step} and \textit{stepL}. They are both presented below:

\begin{verbatim}
step(x, y : list; a, b : char)
\end{verbatim}

\begin{verbatim}
stepL(x : list; a : char)
\end{verbatim}

(a) Advance State

(b) Advance to Same State

Figure 2.4: Rules for Simulating FSA

It should be immediately evident that these achieve the desired result. We can now explain why we require an extra character on the end of our input string - at each point, our rules match both the current character and a later one. Without an extra character on the end, the last character of our string would never be matched, as there would be no suitable successor vertex to match with. It is safe to use this fix, as now the ‘X’ vertex on the end will be ignored as the final character would have been.

The overall complexity of this program can be seen to be at worst \(O(|y|)\), where \(y \in \Sigma^*\) is the input string; this is because every match takes constant time, as all vertices matched are either rooted or adjacent to a root node, and degrees of vertices are constant-bounded by \(|\Sigma|\). This is clearly the best complexity we can hope for - anything better would be unable to read all of \(y\) in the time it runs! We have therefore demonstrated an example of how root nodes can nonintrusively improve performance of graph programs, alleviating the burden of subgraph matching by exploiting locality.
2.3 DPDAs

Encouraged by the elegance and success of our FSA simulation program, we now turn our attention to deterministic pushdown automata (DPDAs). A DPDA is defined as a 7-tuple of the form:

\[ M = (Q, \Sigma, \Gamma, q_0, Z_0, A, \delta) \]  

where

- \( \Gamma \) is the finite stack alphabet;
- \( Z_0 \in \Gamma \) is the initial stack symbol;
- \( \delta : Q \times (\Sigma \cup \{\lambda\}) \times \Gamma \to Q \times \Gamma^* \) is the transition function;
- \( Q, \Sigma, q_0 \) and \( A \) are as they were for finite state automata.

At any point throughout execution, a pushdown automata maintains a stack of values in the stack alphabet \( \Gamma \), initially only containing \( Z_0 \). Our new transition function now not only must advance us to a new state, but must read the top value of the stack and place some number of new values onto it. This is represented by the new \( \Gamma \) argument and the augmented \( Q \times \Gamma^* \) return type.

Execution of a DPDA proceeds much in the same way as an FSA, but with the additional complexity of maintaining a stack of values in \( \Gamma \). Our representation of a DPDA in graphical form must now include, on each edge, the value of the stack to be popped off, and the new string of values to push on in its place. This will be represented by more expressive labels for edges in our host graph, to be explored further later.

An annoying issue becomes that of the representation of our stack in GP2. Ideally, we would like to use another linked list much like our input string; however, we quickly meet the issue of variably large numbers of values being pushed onto the stack at once. We cannot represent every such case easily with a finite number of rules, so we must be a bit more creative. Our approach will be to note that all DPDAs can be equivalently expressed with rules that do one of three things:

- Pop a character off the stack and push on two new characters;
- Pop a character off the stack and push on one new character;
- Pop a character off the stack and push on nothing.

This is evident since we can replace any transition that adds \( y \in \Sigma^* \) onto the stack where \( |y| > 2 \) with \( |y| \) transitions that add on each value in \( y \) consecutively, popping off the previous value and pushing it with the next character after. As each \( |y| \) is constant, we retain a constant overhead for any modified transition and our complexity remains the same. Hence, we will assume our DPDAs are of this form.
We require our host graph to contain the following components:

- A linked list of blue-coloured vertices representing the input string with "X" as an appended suffix, where the first character is rooted;
- A linked list of red-coloured vertices representing the starting stack, initially just a rooted vertex labelled "Z" and a successor labelled "X";
- A DPDA graph, identical to an FSA graph, but with edges labelled $\sigma : \alpha : \beta$ where $\sigma \in \Sigma$ is to be read in the input string, $\alpha \in \Gamma$ is to be popped off the stack and $\beta \in \Gamma^*$ is to be pushed on, where $|\beta| \leq 2$.

We will represent an empty $\beta$ with 0, and insist that $\Gamma \subseteq \text{Char}$, where $\text{Char}$ is the character type in GP2. The colon operator used for DPDA graph edges represents list concatenation.

Below is an example DPDA host graph, to cement the concepts we have illustrated.

![Figure 2.5: An Example DPDA Host Graph](image)

The edge label $a : x : y : v$ will read $a$, pop off $x$ and push $y$ then $v$. The label $b : y : 0$ will read $b$ and pop off $y$ without pushing anything.

We require three cases for each possible edge label as per our prior restrictions, bringing us to a total of six rules. The program is as follows:

```
Main = {step0, step1, step2, step0L, step1L, step2L}!
```

![Figure 2.6: GP2 DPDA Program](image)

The rules `step0`, `step1` and `step2` handle cases of pushing zero, one or two values onto the stack while advancing to a new state and `step0L`, `step1L` and `step2L` do the same for looping back to the same state.
We only show the first three, as the rest are easily discerned.

\begin{figure}[h]
\centering
\begin{tabular}{|c|c|}
\hline
\textbf{step0} & \textbf{step1} \\
\hline
\textbf{step2} & \\
\hline
\end{tabular}
\caption{Rules for Simulating FSA}
\end{figure}

The upper two rules may appear to both match some overlapping cases; one could imagine \( n \) in \textit{step1} unifying with the 0 that \textit{step0} looks for in its edge label. However, note that the type of \( n \) is \texttt{Char}, and so will not do this. Hence, the rules are all disjoint.

Similarly to the FSA complexity, we can easily see that the complexity for this program is \( O(|y|) \), where \( y \in \Sigma^* \) is the input string. We can see this due to bounded degree and the earlier arguments about FSA complexity.
Chapter 3

Efficient Topological Sort

3.1 Problem Description

A binary relation $<$ is a (strict) linear order on a set $X$ iff for all $a, b, c$ in $X$, we have:

1. antisymmetry: $a < b \Rightarrow \neg (b < a)$;
2. transitivity: $a < b \land b < c \Rightarrow a < c$;
3. connex property: $a < b \lor b < a$.

A topological sort of a directed graph is a linear ordering of its vertices, such that for every edge $uv$ from vertex $u$ to vertex $v$, $u$ comes before $v$ in the ordering. A topological ordering is possible if and only if the graph has no directed cycles, that is, if it is a directed acyclic graph (DAG).

We shall be interested only in simple DAGs: $G = (V, E)$ where $V$ is a finite set of nodes and $E \subseteq V \times V$ is a set of directed edges. In fact, the transitive closure of $E$ is exactly a strict order on the set $V$, where a strict order is just a linear order, potentially without the connex property. Assuming we have the constant time function $Marked : V \rightarrow \text{Bool}$, then we can perform a topological sort of a simple DAG in linear time (with careful storage/lookup of edges). Calling $Visit$ with the root will perform a topological sort, leaving the output in some linked list $L$:

\[
\text{Visit}(n):
\]
\[
\quad \text{if not Marked}(n):
\]
\[
\quad \quad \text{for all } m \text{ in } V \text{ s.t. } (n, m) \text{ in } E
\]
\[
\quad \quad \quad \text{Visit}(m)
\]
\[
\quad \quad \text{Mark}(n)
\]
\[
\quad \quad \text{L}.\text{Prepend}(n)
\]

Figure 3.1: Pesudo Code Topological Sort
3.2 GP2 Implementation

The previous algorithm is a DFS with a single entry point, thus we can mimic the strategy used to perform 2-colouring [PB12, p. 9-11], producing a small, fast program.

```
Main = Sort; Convert
Sort = init; (visit!; back0)!; back1
Convert = (cleanup!; walk0)!; cleanup!; walk1
```

Figure 3.2: GP2 Topological Sort Program

We are assuming our input graph is a simple non-empty connected DAG, where the node labels themselves are the data we wish to order. Such graphs will always have exactly one root node, and as such, we will expect that the input graph has a GP2 style root marking on said node. We shall also assume the input graph is not coloured or marked.

Figure 3.3: GP2 Sorting Rules
The sort procedure works by first initialising a list that we shall use to point at nodes in the original graph as we choose an ordering. We then attempt to “visit” as many nodes as we can. We do this by walking arbitrarily down the graph, dashing the edges we use as we go. Once we have reached a leaf node, we then walk back up one step, and note the node by linking it to our auxiliary list we made, with the red root node. We then proceed to attempt to walk again, etc. We are performing exactly a depth first search of the graph.

The purpose of the conversion procedure is to delete the original graph and move the labels to our auxiliary linked list structure. A full example has been included in the next section.
3.3 Complexity Results

We claim, without formal proof, that the runtime performance of this program is linear time with respect to the number of input nodes in the host graph, provided that the nodes of the graph are of bounded degree, and the node labels are of bounded size. We have included an example execution, showing the state of the graph before execution, after sorting, and finally after cleanup.

![Input Graph](image1)
![Intermediate Graph](image2)
![Output Graph](image3)

Figure 3.5: Example Topological Sort

Getting from the input graph to the intermediate graph takes linear time with respect to the number of nodes in the input graph, and similarly, converting the intermediate graph to the output graph takes linear time. This is because each node is visited only a constant number of times, and the degree of each node is bounded (which means rule application is constant time and there can only ever be a constant bounded number of edges to delete when applying the `cleanup` rule as long as possible).
We have provided empirical benchmarking results on random simple DAGs of bounded degree with bounded integer labels. Details of the benchmarking system can be found in the appendix.

Figure 3.6: Topological Sort Benchmark
Chapter 4

Efficient Red-Black Trees

4.1 Problem Description

In general, binary search trees perform very poorly on ordered data, with operations taking up to $O(n)$ time, due to the tree potentially becoming unbalanced. What we’d like to do is to keep each tree approximately balanced, so that no operation takes more than $O(\log(n))$ time [Oka98, p. 24]. Red-black trees are one of the most popular ways of doing this [GS78].

A red-black tree is a binary search tree (where the leaf nodes are exactly the empty nodes) in which every node is coloured either red or black, with the following invariants [Oka98, p. 24-25]:

1. no red node has a red child;
2. every path from the root to an empty node contains the same number of black nodes.

These two invariants guarantee that the longest possible path in a red-black tree, one with alternating black and red nodes, is no more than twice as long as the shortest possible path, one with black nodes only. Thus, the maximum depth of a node in a red-black tree of size $n$ is at most $2\log(n + 1)$ [Oka98, p. 25].

4.2 Functional RBTs

We first present a modified version of Okasaki’s Haskell algorithm [Oka98, p. 28], such that double insertion is no longer permitted, and the type signature of balance has been modified in order to simplify translation to GP2. As shown in [Oka98, p. 25-27], the invariants of a RBT hold, so the insert operation has time complexity $O(\log(n))$, because the depth of the tree is at most $2\log(n + 1)$. 
data Color = R | B

data Tree e = E | T Color (Tree e) e (Tree e)

empty :: Tree e
empty = E

insert :: Ord e => e -> Tree e -> Tree e
insert x t = mb $ ins t
where
  ins E = T R E x E
  ins (T color a y b)
  | x < y = balance (T color (ins a) y b)
  | x > y = balance (T color a y (ins b))
  mb (T _ a y b) = T B a y b

balance :: Tree e -> Tree e
balance (T B (T R (T R a x b) y c) z d)
  = T R (T B a x b) y (T B c z d)
balance (T B (T R a x (T R b y c)) z d)
  = T R (T B a x b) y (T B c z d)
balance (T B a x (T R (T R b y c) z d))
  = T R (T B a x b) y (T B c z d)
balance (T B a x (T R b y (T R c z d)))
  = T R (T B a x b) y (T B c z d)
balance t = t

Figure 4.1: Haskell Implementation

4.3 GP2 Implementation

4.3.1 Host Graphs

Before we dive into an implementation, it remains to decide how to encode a RBT (red-black tree) as a GP2 host graph. Rather than using GP2’s native colouring mechanism, it is more convenient to use loops to denote node colour:

![Node Colourings](image)

(a) Black Node  
(b) Red Node

Figure 4.2: Node Colourings
It will be convenient to encode the black “nil” nodes ($E$ in the Haskell implementation) within the GP2 host graph, rather than omitting them and making them implicit, as this allows us to express our rules much more elegantly. We will reserve the label $-1$ for “nil” nodes, and require that elements inserted into the RBT to be natural numbers, that is, non-nil nodes shall be labelled from the set $\{0, 1, 2, \ldots\}$. For example, in our encoding, the empty RBT looks like:

![Figure 4.3: Trivial RBT](image)

Finally, to avoid additional where clauses in our matching, we shall dash the edge joining a parent to its left child, and not dash the edge for the right child. For example, the RBT containing exactly one non-nil node $1$ is:

![Figure 4.4: Singleton RBT](image)

### 4.3.2 Program Outline

The program we present will insert elements into the RBT from a linked list provided with an initial RBT to use. We will colour the nodes from the linked list green in order to distinguish them. That is, host graphs will contain an initial RBT and a green linked list.

For example, given the host graph containing the empty RBT and the elements $\langle 1, 2, 3 \rangle$ to insert is, we can execute the program, and produce the resultant RBT.
4.3.3 Full Program

We now present the GP2 program.

```plaintext
Main = (Search; Insert; Balance)!
Search = {s0, s1}!
Insert = try i0 else i1
Balance = ({r0, r1}; {b0, b1, b2, b3}!); mb!
```

The entry point is the `Main` procedure is the repeated 3-step process of `Search`, `Insert`, and `Balance`. This will continually insert elements from the linked list into the RBT until either all have been inserted, or until there’s a conflict, that is, the candidate for insertion already exists in the tree. In the next three subsections, we shall give details of each of the procedures and rules.

4.3.4 Search Procedure

The first thing we need to do is to search for the leaf node at which we want to insert, performed by the `Search` procedure. The two rules s0 and s1 correspond to the branching logic of the Haskell function `ins`.

We give a graphical printing of the s0 and s1 rules.
where $x < y$

**Figure 4.7: Search Rule 0**

where $x > y$

**Figure 4.8: Search Rule 1**

### 4.3.5 Insert Procedure

If and only if *Search* has completed successfully, then the *Insert* procedure will be able to proceed (due to the matching against the label $-1$). The two rules $i0$ and $i1$ correspond to the base case of the Haskell *ins* function, replacing the selected leaf node with a new red node with “nil” children.

In GP2, we need two rules rather than one because we must handle management of the input linked list. If there are more items to insert, then $s0$ will be matched, popping the current element from the head of the list, otherwise, $s1$ will be matched, removing the input list after insertion.
It is always guaranteed that one of these rules will be matched if the Search was able to complete successfully. Following insertion, we will then need to perform balancing, which will always move our root node back to the root of the tree.

### 4.3.6 Balance Procedure

Recall the definition of the Balance procedure.

\[
\text{Balance} = (\{r0, r1\}; \{b0, b1, b2, b3\})!; \text{mb}!
\]

The recursion from the ins Haskell function is captured by the repeated application of moving up the tree and then performing balancing. In particular, we apply \(r0\) or \(r1\) first, and then attempt to apply one of the balancing rules, analogous to the Haskell balance function: \(b0, b1, b2, b3\). Either exactly one of these rules will be applied, or none of them.
Figure 4.12: Return Rule 0

Figure 4.13: Return Rule 1

Figure 4.14: Balancing Rule 0
Figure 4.15: Balancing Rule 1

Figure 4.16: Balancing Rule 2
Once we have finished balancing, we will ensure the root node is coloured black, just like in the \textit{mb} Haskell function, with the \textit{mb} rule. This rule will be applied zero times if the node is already black (since no match will be found), and will be applied exactly once if the node is red (since it will be matched once, and then no more times after application).
4.3.7 Continued Example

In subsection 4.3.2, we introduced an example where we inserted \(\langle 1, 2, 3 \rangle\) into a RBT. We now continue this example, by inserting \(\langle 5, 4 \rangle\).

Executing our program on the above host graph will yield the following graph:

![Figure 4.19: Input Graph Example 2](image)

![Figure 4.20: Example Output Graph 2](image)
4.4 Complexity Results

Because the height of an $n$ element RBT is $O(\log n)$, and our rule matching is constant time, insertion $i$ to the tree will take $O(\log i)$ time. Hence, the complexity of the entire program’s execution on an $n$-element list will be:

$$O \left( \sum_{i=0}^{n} \log(i) \right) = O(n \log(n)) \quad (4.1)$$

The above complexity result is proved in Appendix B.

We have provided empirical evidence by generating random host graphs containing a list of integers of bounded size, and an empty initial RBT for the elements to be inserted into.

![Program Benchmark](image)

Figure 4.21: RBT Benchmark
Chapter 5

Program Refinement

5.1 Motivation

It is now high time to turn to the second aim of this project - the refinement of unrooted programs by their rooted counterparts. We shall understand refinement to mean an increase in determinism. Hence, a refinement of a program should only be able to enact a strict subset of the behaviours of said program. This definition is evidently too vague as it stands, and needs more investigation to be of any use to us. Of course, example is often the best teacher; it is in our interest to find a terribly simple example and explore some rooted refinements of it, showing concretely what a rooted refinement should really look like. After this exercise, we will be privy to an intuition around this problem that can be formalised into solid theory.

To find our example, we pose a question: what is the simplest graph program? There are various programs that might be considered “simple”. Consider the programs $Main = \text{skip}$ or $Main = \{\}$, which terminate immediately on every input, the second program merely failing in all cases. These programs, however, are too trivial to be of interest. Perhaps we need to rephrase the question: what is the simplest interesting graph program? There are a few in the standard literature, such as the program that computes the transitive closure of a graph, or one that reverses the direction of every edge in a graph. Again, our question needs some more tuning. What is the simplest nondeterministic program for which there exists a deterministic refinement, with a linear number of steps with respect to the size of the input graph? Reversing edges in a digraph looks like an attractive case to start with. An obvious nondeterministic algorithm for this is as follows:

\[
S := \text{all proper edges in their original orientation}
\]

while $S$ is non-empty:

reverse an edge from $S$

Figure 5.1: Pesudo Code Reversal
Clearly this will reverse every proper edge in the graph exactly once, and will always terminate since we are guaranteed to have only finitely many edges in a finite graph. What is the time complexity of this program? Linear? Clearly this can’t be done in better than linear time, since the number of edges that must be reversed is a linear function of the graph’s size! Can this be performed in linear time then? We will return to this question shortly.

Before we continue, we must say that, within the entirety of this chapter, we are restricting the input space to GP2 graphs that are fully labelled, contain no root nodes, no marked nodes, no coloured nodes, no dashed edges, and no coloured edges. This helps us to reduce the number of rules we need to write, or to reserve features for internal use.

We now implement our edge reversal program in GP2. We can reverse any proper edge not yet reversed, noting which ones we have already reversed, by dashing them. Once we have reversed every proper edge, it remains to undash every proper edge.

Main = reverse!; undash!

Figure 5.2: GP2 Reversal Program

![Reverse & Dash Rule](image)

reverse(x, y, z : list)

![Undash Rule](image)

undash(x, y, z : list)

Figure 5.3: GP2 Reversal Rules

What is the time complexity of this program? Unfortunately, the answer is not linear. This is due to the nature of rule application in GP2; every time we wish to apply our reverse or undash rules, we must search for a match in the graph. In the case of our rules, this process has time complexity $O(n^2)$, where $n$ is the size of the graph. This is because we must search over every node in the graph for an edge connecting with another node, and the maximum degree of any node in the graph is $n$. Unfortunately, this means our program must run in $O(n^3)$ time - a long way from the linear time algorithm we were hoping for!

There are two observations to make here:

1. Oftentimes, graph algorithms are applied to graphs of bounded degree. In such situations, the time taken to look through all adjacent nodes can be considered a constant factor, giving us quadratic time complexity in practice.
2. We’re not yet using root nodes. Are these not designed to “improve time complexity”? Well, yes, they are, but the process of adding root nodes to a program is not simple. For starters, we must ensure that we only ever have a bounded number of root nodes at any time in our working graph. We must also devise a way of using them, for each program, that actually improves the time complexity!

For the remainder of this chapter, this case study will serve as our running example. We will investigate how we can use root nodes to refine this program, improving its time complexity by a tremendous factor.

5.2 Refinement I

In order to make progress, we shall restrict the graph we are willing to accept to only non-empty linked lists. Non-empty linked lists are defined by following properties:

1. there is exactly one node with incoming degree 0;
2. the incoming degree of every node is at most 1;
3. the outgoing degree of every node is at most 1.

We first check if restricting ourselves to only non-empty linked lists has improved the time complexity of our existing program. Clearly, all our nodes now have bounded degree (2), however the number of edges remains a linear function of the size of the graph, so this restriction alone does not improve time complexity of this algorithm any more than the more general restriction of bounding the degree of nodes. All is not lost, however. It should be possible, in theory, to traverse the graph without needing any auxiliary memory or backtracing; by starting from the root node and travelling along every edge in sequence, we will clearly visit every node in the graph.

If we can use root nodes to track this traversal, then we can limit the search space for each rule application, so that we can apply each rule in constant time! Consider the following refinement:

```
Main = root; reverseR! undashR!; unroot
```

Figure 5.4: Rooted Linked List Reversal

The program’s operation is simple, and almost identical to our unrooted example from before. The first rule `root` simply adds a root to the beginning of the list; it can’t add this anywhere else, due to the fact our to-be-rooted node is not an interface node and would violate the dangling condition if anything connected to it. This is a common design pattern we employ to improve elegance of graph programs.

After this, we simply step through each node of the list from top to bottom by applying `reverseR`, reversing and dashing each node as we go. This act of ‘stepping through’ the list is simulated by the root node - at each application of the rule, the root node advances
one down the list, and will be matched again in the next rule application. When we cannot do this any more, we have reached the list’s end, and undash nodes from bottom to top via another list traversal in \textit{undashR}. Finally, we remove our root with \textit{unroot}.

Note that we could have avoided using \textit{undashR} at all, instead just calling \textit{unroot} immediately after \textit{reverseR} and not dashing edges in \textit{reverseR}. This, however, would not have allowed us to note how our program refines the original unrooted one. Note that if we ignored which nodes were rooted, every possible execution of our program exactly corresponds to a possible execution of our unrooted one. Our program adds nothing but more determinism, restricting the order in which edges are reversed and undashed from any order to one possible sequence. This captures our concept of refinement, being defined as an increase in determinism.

To this end, in the context of showing refinement, we prefer rooted programs that correspond closely to their unrooted counterpart, altering at most the range of possible paths of execution. We are lucky that this program translates immediately to a rooted one, and the refinement is clear due to the restricted data structure.

\begin{figure}[h]
\centering
\begin{tabular}{ll}
root(x, y, z : list) & unroot(x : list) \\
\begin{tikzpicture}
  \node (x) at (0,0) [shape=circle,draw] {x};
  \node (y) at (1,1) [shape=circle,draw] {y};
  \node (z) at (1,-1) [shape=circle,draw] {z};
  \draw (x) -- (y);
  \draw (y) -- (z);
\end{tikzpicture} & \begin{tikzpicture}
  \node (x) at (0,0) [shape=circle,draw] {x};
\end{tikzpicture} \\
\end{tabular}
\begin{tabular}{ll}
\hspace{1cm} & \hspace{1cm} \\
⇒ & ⇒
\end{tabular}
\begin{tabular}{ll}
(a) Add Root & (b) Remove Root \\
\rotatebox{90}{\begin{tikzpicture}
  \node (x) at (0,0) [shape=circle,draw] {x};
  \node (y) at (1,1) [shape=circle,draw] {y};
  \node (z) at (1,-1) [shape=circle,draw] {z};
  \draw (x) -- (y);
  \draw (y) -- (z);
\end{tikzpicture}} & \begin{tikzpicture}
  \node (x) at (0,0) [shape=circle,draw] {x};
\end{tikzpicture} \\
\hspace{1cm} & \hspace{1cm} \\
\hspace{1cm} & \hspace{1cm} \\
\rotatebox{90}{\begin{tikzpicture}
  \node (x) at (0,0) [shape=circle,draw] {x};
  \node (y) at (1,1) [shape=circle,draw] {y};
  \node (z) at (1,-1) [shape=circle,draw] {z};
  \draw (x) -- (y);
  \draw (y) -- (z);
\end{tikzpicture}} & \begin{tikzpicture}
  \node (x) at (0,0) [shape=circle,draw] {x};
\end{tikzpicture} \\
\hspace{1cm} & \hspace{1cm} \\
\hspace{1cm} & \hspace{1cm} \\
\rotatebox{90}{\begin{tikzpicture}
  \node (x) at (0,0) [shape=circle,draw] {x};
  \node (y) at (1,1) [shape=circle,draw] {y};
  \node (z) at (1,-1) [shape=circle,draw] {z};
  \draw (x) -- (y);
  \draw (y) -- (z);
\end{tikzpicture}} & \begin{tikzpicture}
  \node (x) at (0,0) [shape=circle,draw] {x};
\end{tikzpicture} \\
\hspace{1cm} & \hspace{1cm} \\
\hspace{1cm} & \hspace{1cm} \\
\rotatebox{90}{\begin{tikzpicture}
  \node (x) at (0,0) [shape=circle,draw] {x};
  \node (y) at (1,1) [shape=circle,draw] {y};
  \node (z) at (1,-1) [shape=circle,draw] {z};
  \draw (x) -- (y);
  \draw (y) -- (z);
\end{tikzpicture}} & \begin{tikzpicture}
  \node (x) at (0,0) [shape=circle,draw] {x};
\end{tikzpicture} \\
\hspace{1cm} & \hspace{1cm} \\
\hspace{1cm} & \hspace{1cm} \\
\rotatebox{90}{\begin{tikzpicture}
  \node (x) at (0,0) [shape=circle,draw] {x};
  \node (y) at (1,1) [shape=circle,draw] {y};
  \node (z) at (1,-1) [shape=circle,draw] {z};
  \draw (x) -- (y);
  \draw (y) -- (z);
\end{tikzpicture}} & \begin{tikzpicture}
  \node (x) at (0,0) [shape=circle,draw] {x};
\end{tikzpicture} \\
\hspace{1cm} & \hspace{1cm} \\
\hspace{1cm} & \hspace{1cm} \\
\rotatebox{90}{\begin{tikzpicture}
  \node (x) at (0,0) [shape=circle,draw] {x};
  \node (y) at (1,1) [shape=circle,draw] {y};
  \node (z) at (1,-1) [shape=circle,draw] {z};
  \draw (x) -- (y);
  \draw (y) -- (z);
\end{tikzpicture}} & \begin{tikzpicture}
  \node (x) at (0,0) [shape=circle,draw] {x};
\end{tikzpicture} \\
\hspace{1cm} & \hspace{1cm} \\
\hspace{1cm} & \hspace{1cm} \\
\rotatebox{90}{\begin{tikzpicture}
  \node (x) at (0,0) [shape=circle,draw] {x};
  \node (y) at (1,1) [shape=circle,draw] {y};
  \node (z) at (1,-1) [shape=circle,draw] {z};
  \draw (x) -- (y);
  \draw (y) -- (z);
\end{tikzpicture}} & \begin{tikzpicture}
  \node (x) at (0,0) [shape=circle,draw] {x};
\end{tikzpicture} \\
\hspace{1cm} & \hspace{1cm} \\
\hspace{1cm} & \hspace{1cm} \\
\rotatebox{90}{\begin{tikzpicture}
  \node (x) at (0,0) [shape=circle,draw] {x};
  \node (y) at (1,1) [shape=circle,draw] {y};
  \node (z) at (1,-1) [shape=circle,draw] {z};
  \draw (x) -- (y);
  \draw (y) -- (z);
\end{tikzpicture}} & \begin{tikzpicture}
  \node (x) at (0,0) [shape=circle,draw] {x};
\end{tikzpicture} \\
\hspace{1cm} & \hspace{1cm} \\
\hspace{1cm} & \hspace{1cm} \\
\rotatebox{90}{\begin{tikzpicture}
  \node (x) at (0,0) [shape=circle,draw] {x};
  \node (y) at (1,1) [shape=circle,draw] {y};
  \node (z) at (1,-1) [shape=circle,draw] {z};
  \draw (x) -- (y);
  \draw (y) -- (z);
\end{tikzpicture}} & \begin{tikzpicture}
  \node (x) at (0,0) [shape=circle,draw] {x};
\end{tikzpicture} \\
\end{tabular}
\end{figure}

\textbf{Figure 5.5: Rules for Rooted Linked List Reversal}
5.3 Refinement II

How far can we take this? What if we lifted our restriction to non-empty trees rather than linked lists?

Our graph will be defined to represent a tree exactly when for any two nodes \( v_1, v_2 \in V \), we have exactly one path from \( v_1 \) to \( v_2 \) and there is exactly one node with no incoming edges. We call this particular node the tree’s “root node”.

We stop to consider what execution paths of the original graph reversal program we can emulate in trees. Because our rooted program will only reverse edges nearby to a root node, we will need a way to propagate our root node to eventually reach every edge in the tree. Without doing this, we will have to reverse at least one edge that is not within the neighbourhood of a root node, requiring a rule match of at least linear time. Hence, we should only consider execution paths of graph reversal that reverse edges in the order of some local traversal of the graph. For linked lists, this was easy - we simply considered paths that reversed edges from the list’s start to its end, then undashed in the reverse order. We can do similarly with trees, by considering paths that form a depth-first traversal.

Our program is as follows:

```
Main = rootT; Reverse; unrootT
Reverse = (steprevT!; backundashT)!
```

Figure 5.6: Rooted Tree Reversal

This program can be seen to adopt the approach of a depth-first search, where we step deeper into the tree with \( \text{steprevT} \). The fact that we dash the previous edge actually turns out to be of greater benefit than before, as it stops us from stepping back somewhere we’ve already been to. We apply \( \text{steprevT} \) as much as we can, taking us to a leaf node of the tree, reversing and dashing nodes along the way. Once we reach a leaf node, we step back once with \( \text{backundashT} \), undashing the edge; as it is reversed, we will now not step in this direction again if we apply \( \text{steprevT} \). Hence, this sequence of traversing as deeply as we can followed by stepping back once when there remains nothing new to explore is a sufficient instantiation of a depth-first traversal.

We also note the complexity of this program to be indeed \( O(|V|) \), where \( |V| \) is the number of vertices in our graph; we only visit each node twice, once when traversing downwards and once when stepping back upwards. This invariant is ensured due to reversed edges leaving a root node being impossible for \( \text{steprevT} \) to match.

It remains to consider how this relates to the original unrooted reversal program. It should be evident that our depth-first traversal reaches every edge in our program, and that each one is first reversed and dashed, then undashed afterwards. Hence, our program will always enact an execution of the original program - a notion that we remind ourselves is the exact definition of refinement we seek.
5.4 Refinement III

We finish by lift our restriction from trees to forests with a bounded number of trees. A forest is a weakening of the definition of a tree, requiring only that for any \( v \in V \), there exists a node with no incoming edges \( v_r \in V \) such that there is exactly one directed path from \( v_r \) to \( v \). It is easy to verify that any connected component of a forest is a tree by our prior definition.

Again, to produce a rooted program, we must consider an execution path with sufficient locality to benefit from adding root nodes. Our remark about forests being a graph with tree components was not purely a quaint aside - indeed, it is this property that we will exploit, reaping the fruits of our efforts to optimise reversal of tree graphs.

As a matter of fact, a possible program could be as follows, using only the rules from the tree reversal program we already have available:

```
Main = rootT! Reverse; unrootT!
Reverse = (steprevT!; backundashT)!
```

Figure 5.8: Rooted Forest Reversal
This program will convert every tree root to a root node. Note that as each component has exactly one tree root, no two root nodes will be in the same component. Hence, we can simply run Reverse, which notably will reverse several tree components concurrently in an arbitrary interleaved order. The rules steprevT and backundashT will simply choose some root node to match against, and progress the depth-first traversal by one rule application on the component containing that root node. Eventually, all depth-first traversals will terminate of all components, at which point Reverse will no longer be applicable and all rooted nodes will be unrooted with unrootT.

The complexity of this program is quite obviously \( O(|V| \times C) \), where \( C \) is the number of connected components in the graph. However, as per our original assumption of a bounded number of components, this will simply be a constant. Hence, we have again achieved linear time, elegantly generalising our tree program by merely applying rootT and unrootT as many times as possible.

### 5.5 Refinement of List Sorting

We end this chapter by considering a different class of program to highlight the limitations of our approach to refinement, namely sorting lists. Suppose we have an input graph that is a linked list where the nodes are labelled with integers, and the edges are unlabelled. Then, we can clearly sort this list using a nondeterministic program:

```
Main = sort!
```

#### (a) Program Specification

```
sort(x, y : int)
⇒
```

where \( x > y \)

#### (b) Sort Rule

Figure 5.9: GP2 Sorting Program

Clearly this program will swap elements until there are no pairwise elements out of order, and this is exactly the postcondition for sorting, and so this program is partially correct. It remains to show that it always terminates. Termination is clear from the fact that for a list of finite length \( n \), at most \( n^2 \) comparisons will be made, and progress is guaranteed due to the as long as possible semantics.
Given our nondeterministic sorting program, we wish to refine it to be deterministic. We give an intuitive refinement to inserting sort for linked lists with at least 2 elements, based on the well-known insertion sort algorithm.

Main = root; walk!; Sort!; unroot
Sort = step; swap!; back!

(a) Program Specification

(root(x, y : int)

unroot(x : int)

walk(x, y : int)

step(x, y : int)

swap(x, y : int)

where x > y

unroot(x : int)

(x)

y

⇒

0

x

y

⇒

0

x

1

y

1

y

1

y

1

y

⇒

0

x

y

1

y

1

y

1

y

⇒

0

x

y

1

y

1

y

1

y

⇒

0

x

y

1

y

1

y

1

y

⇒

0

x

y

1

y

1

y

1

y

⇒

0

x

y

1

y

1

y

1

y

⇒

0

x

y

1

y

1

y

1

y

⇒

0

x

y

1

y

1

y

1

y

⇒

0

x

y

1

y

1

y

1

y

⇒

0

x

y

1

y

1

y

1

y

⇒

0

x

y

1

y

1

y

1

y

⇒

0

x

y

1

y

1

y

1

y

⇒

0

x

y

1

y

1

y

1

y

⇒

0

x

y

1

y

1

y

1

y

⇒

0

x

y

1

y

1

y

1

y

⇒

0

x

y

1

y

1

y

1

y

⇒

0

x

y

1

y

1

y

1

y

⇒

0

x

y

1

y

1

y

1

y

⇒

0

x

y

1

y

1

y

1

y

⇒

0

x

y

1

y

1

y

1

y

⇒

0

x

y

1

y

1

y

1

y

⇒

0

x

y

1

y

1

y

1

y

⇒

0

x

y

1

y

1

y

1

y

⇒

0

x

y

1

y

1

y

1

y

⇒

0

x

y

1

y

1

y

1

y

⇒

0

x

y

1

y

1

y

1

y

⇒

0

x

y

1

y

1

y

1

y

⇒

0

x

y

1

y

1

y

1

y

⇒

0

x

y

1

y

1

y

1

y

⇒

0

x

y

1

y

1

y

1

y

⇒

0

x

y

1

y

1

y

1

y

⇒

0

x

y

1

y

1

y

1

y

⇒

0

x

y

1

y

1

y

1

y

⇒

0

x

y

1

y

1

y

1

y

⇒

0

x

y

1

y

1

y

1

y

⇒

0

x

y

1

y

1

y

1

y

⇒

0

x

y

1

y

1

y

1

y

⇒

0

x

y

1

y

1

y

1

y

⇒

0

x

y

1

y

1

y

1

y

⇒

0

x

y

1

y

1

y

1

y

⇒

0

x

y

1

y

1

y

1

y

⇒

0

x

y

1

y

1

y

1

y

⇒

0

x

y

1

y

1

y

1

y

⇒

0

x

y

1

y

1

y

1

y

⇒

0

x

y

1

y

1

y

1

y

⇒

0

x

y

1

y

1

y

1

y

⇒

0

x

y

1

y

1

y

1

y

⇒

0

x

y

1

y

1

y

1

y

⇒

0

x

y

1

y

1

y

1

y

⇒

0

x

y

1

y

1

y

1

y

⇒

0

x

y

1

y

1

y

1

y

⇒

0

x

y

1

y

1

y

1

y

⇒

0

x

y

1

y

1

y

1

y

⇒

0

x

y

1

y

1

y

1

y

⇒

0

x

y

1

y

1

y

1

y

⇒

0

x

y

1

y

1

y

1

y

⇒

0

x

y

1

y


We have performed an empirical comparison between the nondeterministic specification and the deterministic refinement. The runtime of the nondeterministic program massively dwarfs that of the deterministic program.

(a) Program Comparison

(b) Deterministic Sorting

Figure 5.11: Sorting Benchmarks
Chapter 6

Conclusion

In this report, we have demonstrated a large body of examples for the usage of root nodes in graph programming challenges, and how this can massively improve runtime performance and complexity. Case studies have included diverse problems, such as FSA simulation, sorting and red-black tree insertion. An important result from this was an enhanced insight into design patterns usable with root nodes, such as depth-first traversals, and how data structures can be equivalently optimized to meet expected performance quota. Empirical results have shown consistently how our rooted programs vastly outperform their unrooted counterparts, thanks to a powerful benchmarking utility we have implemented.

We have also further explored some of the limitations of root nodes as a construct, noting that all of our programs that could take advantage of root nodes have had some element of locality that could be exploited. For instance, FSA simulation could only be improved due to its locality of execution, and our list reversal of forests was only truly linear when a bounded number of connected components was present. This suggests an inherent limitation upon root nodes as a construct, which is to be expected - root nodes simply provide a way to access a local constant area of a graph in constant time. Our commonplace restriction of bounded degree also highlights a second constraint of root nodes, in that they cannot cope well with unbounded degree environments. If this limitation is to be surpassed, a further optimization must be considered.

A natural second phase to this project would be to define a notion of refinement by determinism, as could be used to formalize the root node placement in the examples from Chapter 5.

Overall, we have shown the efficacy of root nodes as an optimization and how easily programs using them may be seen as a refinement of unrooted versions. We hope this will lead on to further work in refinement patterns, along with suggesting further optimizations available to graph programs to compensate for other performance obstacles.
Appendix A

Project Tooling

A.1 Overview

There already exists a small set of tooling for GP2. Most importantly, Bak developed a compiler to C for GP2 programs. [Bak15, ch. 5] Within this work the syntax and semantics of GP2 were extended beyond the original publication [Plu12]. We shall be treating [Bak15, ch. 3-4] as the definition of GP2, using the C compiler as our reference implementation.

Unfortunately, the compiler here quite literally takes a GP2 program, and produces C code. In order to run our programs, we need more. Namely, we'd like to be able to provide an input program and host graph, and have it executed, and the result returned. Due to the low compilation times (GP2 programs are typically small compared to their host graphs), it is feasible that we could create something that looked like an interpreter:

1. compiling the input program to C using Bak’s compiler;
2. compiling the resultant C code to an executable using GCC;
3. running the executable on the host graph.

Details of this are discussed in the GP2I section of the appendix. Our next piece of work was to develop a way to benchmark the performance of GP2 programs. In order to benchmark programs, we needed:

1. a way to run our programs on an input;
2. a way to generate input graphs.

Our first piece of work provides a way to run our programs, however, it remained to build a way to generate input graphs, and build the benchmark software. Details of this can be found in the GP2 Bench section of the appendix.

Finally, the translation of GP2 graphs and rules to LaTeX figures is a manual and slow process. We have automated this process by parsing GP2 graphs and rules, and compiling them to dot and tex files, and in turn compiling the dot files to eps, as detailed in the LaTeX section of the appendix.
A.2 GP2I

As discussed earlier, the purpose of GP2I is to bridge the gap between the GP2 Compiler and actually executing GP2 programs on a given host graph. Due to the low compilation times, we can create something that behaves like an interpreter from the outside, but is in fact using the GP2 Compiler behind the scenes:

1. compiling the input program to C using Bak’s compiler;
2. compiling the resultant C code to an executable using GCC;
3. running the executable on the host graph.

A.2.1 Compiler Modifications

Unfortunately, the existing implementation of the GP2 Compiler reserves a fixed amount of working memory, hard coded within the compiler. We have modified the compiler to accept, as command line arguments, arbitrary values for the maximum number of nodes or edges for which to reserve memory.

<table>
<thead>
<tr>
<th>File</th>
<th>Lines Added</th>
<th>Lines Removed</th>
</tr>
</thead>
<tbody>
<tr>
<td>lexer.l</td>
<td>+2</td>
<td>-2</td>
</tr>
<tr>
<td>parser.y</td>
<td>+2</td>
<td>-1</td>
</tr>
<tr>
<td>genProgram.c</td>
<td>+4</td>
<td>-7</td>
</tr>
<tr>
<td>genProgram.h</td>
<td>+7</td>
<td>-1</td>
</tr>
<tr>
<td>main.c</td>
<td>+81</td>
<td>-42</td>
</tr>
<tr>
<td>parser.y</td>
<td>+5</td>
<td>-5</td>
</tr>
</tbody>
</table>

Figure A.1: Modified Files

Usage:

```
gp2 -p <program_file>
gp2 -r <rule_file>
gp2 -h <host_file>
```

Flags:

- `-c` - Enable graph copying.
- `-d` - Compile program with GCC debugging flags.
- `-p` - Validate a GP 2 program.
- `-r` - Validate a GP 2 rule.
- `-h` - Validate a GP 2 host graph.
- `-l` - Specify root directory of installed files.
- `-o` - Specify directory for generated code and program output.
- `--max-nodes` - Specify maximum number of nodes in a host graph.
- `--max-edges` - Specify maximum number of edges in a host graph.

Figure A.2: New GP2 Usage Docs
A.2.2 Interface Description

Now that we have an updated GP2 Compiler executable, we need to design an interface for our “interpreter”. We’d like to make it as easy to use as possible, and so we decided to use gRPC, a high performance multi-language RPC framework, build on top of HTTP 2 and Google Protocol Buffers. [LLC18] We first define our interface description:

```protobuf
syntax = "proto3";

message Request {
  string program = 1;
  string graph = 2;
}

message Response {
  oneof payload {
    string graph = 1;  // the output graph
    string error = 2;  // an error message
  }
  uint32 time = 3;  // execution time
}

service GP2I {
  rpc Interpret (Request) returns (Response);
}
```

Figure A.3: GP2I Interface Description

Looking at this interface, we can see that we have defined a service, \texttt{GP2I}, with one available procedure, \texttt{Interpret}, which expects a \texttt{Request} from the caller, and will provide a \texttt{Response}. As we can see, the \texttt{Request} must contain an input program and a host graph, and the \texttt{Response} will contain either a result graph, or an error message, and will additionally include the execution time of the GP2 program (excluding compilation time).

A.2.3 gRPC Service Implementation

Given the interface description, the gRPC tooling can generate us server and client code in a wide array of supported languages. We have decided to go with Python, but the choice is totally arbitrary.

The implementation of the server is now trivial, and the semantics have already been described. Our service will also respect environment variables dictating the maximum number of nodes and edges to be passed to the compiler.
A.3 GP2 Bench

GP2 Bench is a benchmarking system that compares the runtime performance of an arbitrary number of programs against various graphs, as generated by an arbitrary graph generation service.

A.3.1 Interface Description

We have already defined an interface description for executing and timing programs on graphs. It remains to define an interface description for the generation of host graphs.

```plaintext
syntax = "proto3";

message Unit {
    //
}

message Graph {
    string graph = 1;
}

service Gen {
    rpc Generate (Unit) returns (stream Graph);
}
```

Figure A.4: Generation Interface Description

We have defined a service, Gen, with one available procedure, Generate, which expects nothing from the caller (more precisely, it expects an instance of Unit, which carries exactly no data), and will provide a stream of Graphs in return.

A.3.2 Generator Implementation

Implementation of the graph generation service is similar to that of the GP2I service. The implementation expects some command line arguments on startup. The generation mode is the only required parameter, with the following modes available:

1. fsa (allows −complete option);
2. cycle (no options);
3. tree (allows −rooted option);
4. grid (allows −rooted option);
5. star (allows −rooted option);
6. llist (no options).
It’s also possible to change the “step”, “minimum”, and “maximum” using environment variables, thus controlling the number of graphs that will be streamed to clients.

### A.3.3 Bench Implementation

GP2 Bench is implemented in Python too, with generated gRPC clients. The implementation expects to be provided with, at runtime:

1. 1 or more file paths of target GP2 programs, via command line arguments;
2. the addresses of the GP2I and Generator services, via environment variables.

It is also possible to modify the number of runs of each graph to each program with an environment variable, the default being 20.

### A.3.4 Example Benchmark

We provide an example output from having setup an instance of the GP2I interpreter with the defaults, an instance of the Generator set to generate 10 Red-Black Tree host graphs, and the benchmark software is provided with the red-black tree program from earlier (and the addresses of the 2 services).

```
10 graphs [00:00, 25.60 graphs/s]
100%|================================|= 200/200 [00:27<00:00, 7.27 runs/s]

| Graph | Success | Time     |
|-------+----------+----------|
| 0     | 100.0%   | 26.8293ms|
| 1     | 100.0%   | 46.83905ms|
| 2     | 100.0%   | 67.7259ms|
| 3     | 100.0%   | 90.4281ms|
| 4     | 100.0%   | 113.0349ms|
| 5     | 100.0%   | 139.1417ms|
| 6     | 100.0%   | 162.2712ms|
| 7     | 100.0%   | 189.43255ms|
| 8     | 100.0%   | 219.4736ms|
| 9     | 100.0%   | 240.15885ms|
```

Figure A.5: Example Benchmark
A.4 \textsc{LaTeX}

The translation of GP2 graphs and rules to \LaTeX\ figures is a manual and slow process, one which we so unhappy with, that we decided to automate this process.

Obviously, the first thing we need, is the ability to parse GP2 rules and graphs. Bak’s work is tightly coupled to the C compiler, so we decided to write our own grammar for GP2 rules in LBNF, so that we could use BNFC to generate a lexer and parser in Haskell.

```
#!/usr/bin/env bash

echo "Running BNFC"
bnfc --haskell --functor --ghc $1.cf
rm Doc${1^}.txt

echo "Running Happy"
happy -gca Par${1^}.y
rm Par${1^}.y

echo "Running Alex"
alex -g Lex${1^}.x
rm Lex${1^}.x
```

Figure A.6: LBNF to Haskell

Our LBNF grammar is more permissive than Bak’s C implementation, however, we believe this to be perfectly acceptable, as we are not trying to build a reference interpreter; we’re simply looking to convert GP2 rules and graphs to \LaTeX\ figures. Since our grammar is symmetric in it’s parsing of the LHS and RHS of rules, and is more permissive than Bak’s the parsing of graphs within rules doubles as a way to parse host graphs. We list the grammar over the page.

```
Parser/rule.cf
Common.hs
Main.hs
Nodes.hs
Presenter.hs
Printer.hs
Processor.hs
Rewriting.hs
```

Figure A.7: Non-generated Code Files

Our automation parses GP2 graphs and rules, and compiles them to dot and tex files, and in turn compiling the dot files to eps. The tex file can then be included in other tex documents, thus rendering the figure.
Figure A.8: rule.cf
module Main where

import Control.Monad
import Data.Text (splitOn, unpack, pack)
import System.Environment (getArgs)
import System.Exit (exitFailure, exitSuccess, exitWith, ExitCode(ExitFailure))
import ErrM
import LexRule (Token)
import qualified AbsRule
import ParRule (myLexer, pRuleDecl, pGraph)
import Common
import Processor (convertRawRule, convertGraph)
import Presenter (presentRule, presentGraph)

type Lexer = String -> [Token]

type Parser a = [Token] -> Err a


rulePipeline :: Lexer -> Parser (AbsRule.RuleDecl a) -> Processor (RawRule a) Rule -> String -> Err Rule
rulePipeline lex par pro str =
  let t = lex str
  in case par t of
    Bad err ->
      fail $ unlines ["Parse failed: " ++ err, "Tokens: " ++ show t]
    Ok (AbsRule.RDec _ n a l r i) ->
      pro (RawRule n a l r i Nothing)
    Ok (AbsRule.RDecW _ n a l r i w) ->
      pro (RawRule n a l r i (Just w))


graphPipeline :: Lexer -> Parser (AbsRule.Graph a) -> Processor (AbsRule.Graph a) ([NodeKey], Graph) -> String -> Err Graph
graphPipeline lex par pro str =
  let t = lex str
  in case par t of
    Bad err ->
      fail $ unlines ["Parse failed: " ++ err, "Tokens: " ++ show t]
    Ok g ->
      case pro g of
        Ok (_, g) -> return g
        Bad err -> fail err

run :: Pipeline a -> Presenter a -> Output -> String -> IO ()
run pln pnt out str =
  case pln str of
    Bad r ->
      putStrLn r
      exitWith (ExitFailure 2)
    Ok d ->
      mapM_ writeF $ pnt d
      exitSuccess

writeF (path, ctnt) =
  let
    p = out ++ "/" ++ path
  in do
    writeFile p ctnt
    putStrLn $ "Written " ++ p

main :: IO ()
main =
  do
    args <- getArgs
    case args of
      
runPipeline ("rule", f, o) -> run (rulePipeline myLexer pRuleDecl convertRawRule) (presentRule $ name f) o
runPipeline ("graph", f, o) -> run (graphPipeline myLexer pGraph (convertGraph [])) (presentGraph $ name f) o

main :: IO ()
main =
  do
    args <- getArgs
    case args of
      ("rule", f, o) -> run (rulePipeline myLexer pRuleDecl convertRawRule) (presentRule $ name f) o
      ("graph", f, o) -> run (graphPipeline myLexer pGraph (convertGraph [])) (presentGraph $ name f) o
    putStrLn $ "Expected a mode followed by exactly one input file and an output directory"

    putStrLn $ "Written " ++ p

Figure A.9: Main.hs

46 of 49
Appendix B

Proofs

B.1 Complexity of RBT Insertion

Definition B.1.1. If \( f, g : \mathbb{N} \to \mathbb{R} \), then \( f(n) = O(g(n)) \) iff:
\[
\exists c \in \mathbb{N}. \exists n_0 \in \mathbb{N}. \forall n \in \mathbb{N}. n \geq n_0 \Rightarrow f(n) \leq cg(n) \tag{B.1}
\]

Definition B.1.2. If \( f, g : \mathbb{N} \to \mathbb{R} \), then \( \Theta(f(n)) = \Theta(g(n)) \) iff:
\[
f(n) = O(g(n)) \land g(n) = O(f(n)) \tag{B.2}
\]

Theorem B.1.1. The factorial logarithms are the same class as the linearithmic functions.
\[
\Theta(\log(n!)) = \Theta(n \log(n)) \tag{B.3}
\]

Proof. It’s easy to see that:
\[
\log(n!) = \log\left(\prod_{i=1}^{n} i\right) = \sum_{i=1}^{n} \log(i) \leq \sum_{i=1}^{n} \log(n) = n \log(n)
\]

Similarly:
\[
\log(n!) = \sum_{i=1}^{n} \log(i) \geq \sum_{i=\lfloor \frac{n}{2} \rfloor}^{n} \log(i) \geq \sum_{i=\lfloor \frac{n}{2} \rfloor}^{n} \log\left(\frac{n}{2}\right) \geq \frac{n}{2} \log\left(\frac{n}{2}\right) = \frac{n}{2} \log(n) - \frac{n}{2} \log(2)
\]

Corollary B.1.1. RBT Insertion has a linearithmic time complexity. That is, the program that inserts \( n \) items into an empty RBT has time complexity \( O(n \log(n)) \).

Proof. Recall that the complexity of the program was \( O(\sum_{i=1}^{n} \log(i)) \). It’s easy to see that this is equal to \( O(\log(\prod_{i=1}^{n} i)) \), which is in turn, by definition, \( O(\log(n!)) \). Finally, application of the above theorem shows this is equal to \( O(n \log(n)) \).
Bibliography


