Improving the GP 2 Compiler

Graham Campbell[1], Jack Romo, and Detlef Plump
Department of Computer Science
University of York, United Kingdom

September 2019

[1]Supported by a Vacation Internship from the Engineering and Physical Sciences Research Council (EPSRC) in the UK.

Contents

List of Figures

Prologue

Abstract

GP 2 is an experimental programming language based on graph transformation rules which aims to facilitate program analysis and verification. Writing efficient programs in such a language is hard because graph matching is expensive, however GP 2 addresses this problem by providing rooted rules which, under mild conditions, can be matched in constant time using the GP 2 to C compiler. In this report, we document various improvements made to the compiler; most notably the introduction of node lists to improve iteration performance for destructive programs, meaning that binary DAG recognition by reduction need only take linear time where the previous implementation required quadratic time.

Foreword

This report summarizes the improvements made to the GP 2 compiler during August and September 2019. This work was, in part, funded by a Vacation Internship of the Engineering and Physical Sciences Research Council (EPSRC) granted to Graham Campbell. The structure of the report is as follows:

1. We start by very briefly introduce algebraic graph transformation, the GP 2 language, and the GP 2 to C compiler.
2. We will then detail our improvements to the compiler, motivating the changes for each change.
3. Next, we give timing results comparing the original implementation to the improvement implementation, commenting on why it is faster and slower on various classes of program.
4. Finally, we will summarize and evaluate our results.

In the appendices, we have provided usage documentation for current the GP 2 Compiler, details of the test suite used to give confidence in the correctness of the new implementation, and details of the benchmarking software including the programs used to generate the input graphs.

Executive Summary

GP 2 is an experimental rule-based programming language based on graph transformation rules which aims to facilitate program analysis and verification. Writing efficient programs in a rule-based language is hard because graph matching is expensive, however GP 2 addresses this problem by providing rooted rules which, under mild conditions, can be matched in constant time by the code generated by the GP 2 to C compiler, to which we have:

1. Introduced node and edge lists to improve iteration performance for destructive programs, allowing the compiler to produce node matching code that skips over an area of deleted nodes in constant time. This allows us to recognise binary DAGs in linear time (given an input graph of bounded degree) by means of a slick reduction algorithm, where in the previous compiler implementation, such a program would have a quadratic worst case time complexity.
2. Fixed parsing input graphs of arbitrary size using Judy arrays. In the previous implementation, one had to specify the maximum number of nodes and edges expected in input host graphs, and the generated code would allocate memory for such a graph on startup, before parsing the input graph. In our updated implementation, it is no longer required to know the maximum input graph size at compile time; we dynamically grow the memory needed during parsing.
3. Added a way to shutdown the generated program as soon as it has written the output graph without cleaning up all the allocated heap memory in a proper way. In general, proper management of the heap memory is a good idea to avoid bugs due to memory leaks or otherwise, however, one might not want to do this in the interest of runtime performance, to allow the process to exit as soon as possible after writing the output.
4. Added a way to ensure matches both reflect root nodes, as well as preserving them. By default, GP 2 will allow a non-root in a rule LHS to match against either a non-root or a root in a host graph, but if we insist on matches reflecting root nodes, matches can only match non-roots against non-roots. This is useful in order to achieve reversibility of derivations and also to have more control over rule application.

We conjecture that the new compiler implementation has worst case time complexity no worse than the original implementation. Moreover, we provide runtime performances of various programs and graph classes as empirical evidence for our conjecture, where timing results differ only by a constant factor. Reduction programs on disconnected graphs are massively improved, changing complexity class, programs that use edge searching heavily, see a runtime performance improvement, such as DFS and transitive closure, and most other programs see a small performance degradation, which we consider worth it for the benefits.

Chapter 1

Introduction

In this chapter, we very briefly review the different approaches to algebraic graph transformation, in order to introduce the language GP 2 and its compiler.

1.1 Graph Transformation

Graph transformation is the rule-based modification of graphs, and is a discipline dating back to the 1970s. There are various approaches to graph transformation, most notably the 'node replacement' [1], 'edge replacement' [2], and 'algebraic' approaches [3] [4], originally developed at the Technical University of Berlin by Ehrig, Pfender, and Schneider [5] [6]. The two major approaches to algebraic graph transformation are the so called 'double pushout' (DPO) approach, and the 'single pushout' (SPO) approach.

Because the DPO approach operates in a structure-preserving manner (rule application in SPO is without an interface graph, so there are no dangling condition checks), this approach is more widely used than the SPO [4] [7]. Moreover, the DPO approach is genuinely local in the sense that each rule application can only modify the local area of the graph in which it is matched, as opposed to SPO, which allows arbitrary implicit edge deletion. More recently, there have been hybrid approaches that attempt to gain the locality of DPO, but the flexibility of SPO, such as the Sesqui-Pushout approach [8] [9], which is compatible with DPO when we have injective matches and linear rules [10].

There are a number of languages and tools, such as AGG [11], GMTE [12], Dactl [13], GP 2 [14], GReAT [15], GROOVE [16], GrGen.Net [17], Henshin [18], PROGRES [19], and PORGY [20]. It is reasonable that a general purpose local graph transformation language should choose the DPO approach with injective matches and linear rules; GP 2 is such a language. Moreover, Habel and Plump show that such languages can be 'computationally complete' [21].

1.2 The GP 2 Language

GP 2 is an experimental non-deterministic rule-based language for problem solving in the domain of graphs, developed at York, the successor of GP [22] [14]. GP 2 is of interest because it has been designed to support formal reasoning on programs [23], with a semantics defined in terms of partially labelled graphs, using the injective DPO approach with linear rules and relabelling [10] [24].

GP 2 programs transform input graphs into output graphs, where graphs are directed and may contain parallel edges and loops. Both nodes and edges are labelled with lists consisting of integers and character strings. This includes the special case of items labelled with the empty list. The principal programming construct in GP 2 consist of conditional graph transformation rules labelled with expressions. Rules operate on 'host graphs' which are labelled with constant values. Formally, the application of a rule to a host graph is defined as a two-stage process in which first the rule is instantiated by replacing all variables with values of the same type, and evaluating all expressions. This yields a standard rule (without expressions) in the DPO approach with relabelling. In the second stage, the instantiated rule is applied to the host graph by constructing two suitable pushouts. The formal semantics of GP 2 is given in the style of Plotkin's structural operational semantics [25]. Inference rules, first given in [14], inductively define a small-step transition relation on configurations. Up-to-date versions can be found in Bak's Thesis [26].

Intuitively, applying a rule $L \Rightarrow R$ to a host graph G works as follows: (1) Replace the variables in L and R with constant values and evaluate the expressions in L and R, to obtain an instantiated rule $\hat{L} \Rightarrow \hat{R}$. (2) Choose a subgraph S of G isomorphic to $\hat{L}$ such that the dangling condition and the rule's application condition are satisfied (see below). (3) Replace S with $\hat{R}$ as follows: numbered nodes stay in place (possibly relabelled), edges and unnumbered nodes of $\hat{L}$ are deleted, and edges and unnumbered nodes of $\hat{R}$ are inserted. In this construction, the 'dangling condition' requires that nodes in S corresponding to unnumbered nodes in $\hat{L}$ (which should be deleted) must not be incident with edges outside S. The rule's application condition is evaluated after variables have been replaced with the corresponding values of $\hat{L}$, and node identifiers of L with the corresponding identifiers of S.

A program consists of declarations of conditional rules and procedures, and exactly one declaration of a main command sequence, which is a distinct procedure named `Main`. Procedures must be non-recursive, they can be seen as macros. We describe GP 2's main control constructs. The call of a rule set $\{r_1, \ldots, r_n\}$ non-deterministically applies one of the rules whose left-hand graph matches a subgraph of the host graph such that the dangling condition and the rule's application condition are satisfied. The call 'fails' if none of the rules is applicable to the host graph. The command `if` C `then` P `else` Q

is executed on a host graph G by first executing C on a copy of G. If this results in a graph, P is executed on the original graph G; otherwise, if C fails, Q is executed on G. The `try` command has a similar effect, except that P is executed on the result of C's execution. The loop command $P!$ executes the body P repeatedly until it fails. When this is the case, $P!$ terminates with the graph on which the body was entered for the last time. The `break` command inside a loop terminates that loop and transfers control to the command following the loop.

Poskitt and Plump have set up the foundations for verification of GP 2 programs [27] [28] [29] [30] using a Hoare-Style [31] system (actually for GP [32] [22]), Hristakiev and Plump have developed static analysis for confluence checking [33] [34], and Bak and Plump have extended the language, adding root nodes [35] [26]. Plump has shown computational completeness [36]. Most recently, Atkinson, Plump, and Stepney have developed a probabilistic extension to GP 2 [37] [38]. Most recently, Campbell, Courtehoute and Plump have been interested in linear time algorithms in GP 2 [39], motivating some of the work in this report. We also build on our earlier work in [40].

1.3 Rooted GP 2 Programs

The bottleneck for efficiently implementing algorithms in a language based on graph transformation rules is the cost of graph matching. In general, to match the left-hand graph L of a rule within a host graph G requires time polynomial in the size of L [35]. As a consequence, linear-time graph algorithms in imperative languages may be slowed down to polynomial time when they are recast as rule-based programs. To speed up matching, GP 2 supports 'rooted' graph transformation where graphs in rules and host graphs are equipped with so-called root nodes, originally developed by Dörr [41]. Roots in rules must match roots in the host graph so that matches are restricted to the neighbourhood of the host graph's roots. We draw root nodes using double circles.

A conditional rule $(L \Rightarrow R, c)$ is 'fast' if (1) each node in L is undirectedly reachable from some root, (2) neither L nor R contain repeated occurrences of list, string or atom variables, and (3) the condition c contains neither an `edge` predicate nor a test $e_1 = e_2$ or $e_1 \,!= e_2$ where both e_1 and e_2 contain a list, string or atom variable. Conditions (2) and (3) will be satisfied by all rules occurring in the following sections; in particular, we neither use the `edge` predicate nor the equality tests.

Theorem 1.1 (Complexity of matching fast rules [35])**.** Rooted graph matching can be implemented to run in constant time for fast rules, provided there are upper bounds on the maximal node degree and the number of roots in host graphs.

1.4 Bak's GP 2 Compiler

Before we discuss our modifications to the GP 2 Compiler[1], we first outline its prior state. The compiler detailed in Bak's Thesis [26] compiled GP 2 programs into C code with a Makefile, which was then compiled by the GCC compiler into an executable.

The original compiler stored a graph as a dynamic array of nodes and another of edges, along with the memorized amount of each element. Internally, these arrays were actually more subtle, consisting of an array of the actual elements and a secondary array of indices that contained nothing, or 'holes'. Focusing on nodes, we term the first array the node array and the second the hole array.

In order to iterate through nodes, a program would simply iterate through all indices between 0 and the largest index holding a node, a value the graph remembers. Each index would have to be checked to ensure it genuinely did hold a node, and was not in fact empty. A pointer could be resolved from the index and the node could be modified as desired.

To delete a node, a program would simply add the given index to the array of holes and set the node's entry in the node array to all zeros. Should the node happen to be the last entry in the array, the number of elements in the array could instead be decreased rather than add another hole. Future iterations through the node array would then skip over this hole in the array, as they checked every entry for being a hole or not. This raised performance issues if a program deleted a large number of nodes, for instance, in a graph reduction algorithm, as the enormous number of holes would make traversing the final smaller graph as slow as the original larger one. A prime example of this is the most obvious program that recognises discrete graphs by reduction:

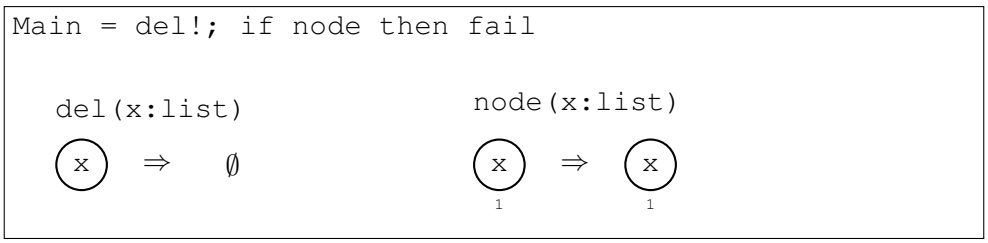
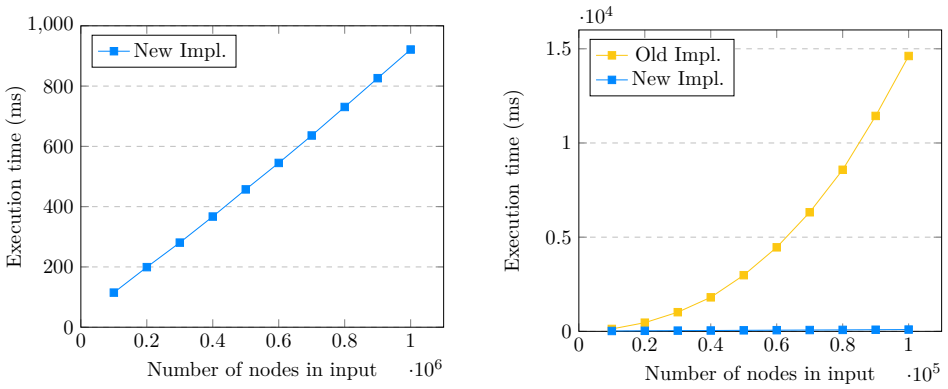


Figure 1.1: GP 2 Program `is-discrete.gp2`

This program should be expected to run in linear time on unmarked, unrooted graphs, as finding an arbitrary node with no further constraints should be a constant-time operation. Alas, each deleted node adds a new hole at the start of the node array, making the program actually take quadratic time due to having to traverse the holes at each rule match.

[1] `https://github.com/UoYCS-plasma/GP2`



(a) New Implementation

(b) Both Implementations

Figure 1.2: Measured Performance of `is-discrete.gp2`

When inserting a new node, one could simply pop off the last element of the holes array to give a free index to work within. If the holes array was empty, the largest used index would have to be incremented and the node placed at the end of the node array. Should the node array be too small, it would be doubled with the `realloc()` C standard library function. The same would be true for the hole array, albeit holes in the hole array would not need to be tracked as only the last element is ever accessed. Arrays would never halve in size should they shrink, to avoid needles memory operations. However, simply reallocating the array raised the possibility of the array changing position in memory to double in size, making any pointers to nodes held while adding a new node invalid. This meant that Bak was indeed required to store indices to nodes instead of pointers, adding extra memory operations to resolve indices every time a node was accessed.

To accommodate root nodes, Bak added a linked list of root nodes to each graph, each entry holding a pointer to a node in the graph's node array. Nodes would contain a flag themselves detailing if they were a root node or not. Root node iteration could then simply be done by traversing this list, a constant time operation should the number of root nodes be constant bounded. Deleting a root node would now require traversing the list as well, to erase the desired entry from said list. These costs are of course a constant time overhead, should the number of root nodes indeed be upper-bounded by a constant in the compiled program.

A node itself would contain a number of entries, including its own index, the number of edges for which it is a target, termed its indegree, the number of edges for which it is a source, termed its outdegree, its incident edges, label, mark and several boolean flags detailing its nature. These flags were stored in separate bool entries in the structure, wasting several bytes where only a bit is used.

The incident edges to a node were stored in a unique manner, with indices

of the first two edges being stored statically as part of the node type itself, and indices of all future incoming and outgoing edges being stored in two dynamically allocated arrays of incoming and outgoing edges, respectively. This would avoid having to allocate arrays for a node should its degree be less than three.

Each edge would contain its own index, its label, mark, and the indices of its source and target nodes. It would, similarly to nodes, hold a boolean flag of whether it had been matched or not yet to prevent overlapping matches.

In order to parse an incoming graph, Bak implemented a graph parser in Bison which would accumulate nodes and edges into a graph as it found them in the input. In order to resolve source and target indices, nodes would be stored in a pre-allocated secondary array, stored at the array index equal to the ID the node was given in the input file. When an edge was then discovered, resolving its source and target could be done by looking up the indices in this array equal to the source and target IDs in the input. Unfortunately, this opened a large number of issues: for instance, the array was unable to change its size, meaning only a fixed number of nodes could be entered into the graph before attempting to store nodes at unallocated indices, resulting in unhandled segmentation faults. Also, should a small number of nodes be given enormous IDs, the parsing stage would have to use a huge amount of memory to create an array able to assign at these indices, making this method both incorrect and inefficient should it have been fixed directly.

Finally, to accommodate programs running within the condition of an if statement, for instance, a stack of states was needed. Bak implemented graph stacks in two varieties: copying the previous graph into a stack to reuse it when unwinding, and storing changes to the graph in the stack to undo them in reverse when unwinding. The former simply stored all the data that a node or edge contained, reconstructing the node or edge as needed as it unwound. Graph copying would simply perform a deep copy of the graph as expected.

1.5 GP 2 Compiler Changes

Since Bak completed his PhD, the GP 2 Compiler implementation has not stood still. We consider the state of the codebase at the end of 1st September 2015 to be the state of the GP 2 compiler, as described by Bak's thesis. This is represented by the `sep-2015` tag on GitHub. Between September 2015 and July 2017 there were various minor fixes and changes made by Bak and Hristakiev.

During the early days of GP 2, a graphical editor[2] was created by Elliot in 2013 [42] in C++. Hristakiev worked to revive this implementation in 2015,

[2] `https://github.com/UoYCS-plasma/GP2-editor`

however the project was abandoned in 2017. Hand has started work on a graph visualizer and visual rule editor[3] to run in the web browser, independently of the C compiler implementation [43]. It is likely that there will eventually be a browser-based GP 2 editor in the future, possibly based on Hand's work.

Most notably, the following changes to the language concrete syntax were completed before the end of December 2015:

1. Replaced the syntax for positioning nodes and edges in rules. This change affects the GP 2 Editor only.
2. String literals can now contain any printable ASCII character.
3. Node and edge identifiers in rules can now start with a digit.
4. Host graph node and edge identifiers now have to be integers.

During the same period, automatic generation of a `Makefile` by the GP 2 compiler was added, and so also the option to validate a single rule as input, just like validating an entire program. It's worth noting that the change to allow string literals to contain any printable ASCII character was actually mistakenly only applied to the program concrete syntax definitions. It was not until March 2017 that the grammar for host graphs was updated too.

Bak's last tweak to the GP 2 Compiler implementation was made in July 2017. Since then, Atkinson, Campbell and Romo have continued to make bug fixes and corrections, without modifying the intended syntax and semantics. Most notably, in July 2018 Romo added the ability to specify as a compiler argument the number of nodes and edges the compiler's generated parser sets aside memory for when parsing host graphs, though this change did not actually become part of the official implementation until February 2019. Previous to that, one had to modify the source code of the compiler itself in order to accept larger input graphs.

During August and September 2019, we have updated the compiler implementation to address the issues described in Section 1.4. We describe the changes in detail in Chapter 2. At time of writing, there exist two branches on GitHub: `legacy` and `master`. The `legacy` branch contains the original GP 2 compiler implementation with support for root reflecting morphisms backported from the new version, which is on the `master` branch.

Finally, it is worth noting that the implementation of the Probabilistic GP 2 (P-GP 2)[4] [38] Compiler remains separate from the GP 2 Compiler, was entirely developed by Atkinson, based on the implementation of GP 2 as it stood after Bak finished making modifications in 2017.

[3] `https://github.com/sdhand/grape`
[4] `https://github.com/UoYCS-plasma/P-GP2`

Chapter 2

Improved Implementation

We now present our modifications to the GP 2 compiler, to overcome the previous issues highlighted. Our first improvements consisted of fixing graph parsing, abandoning dynamic arrays of nodes in favor of Judy arrays. Our next and most significant internal change is employing linked lists for node and edge iteration, allowing one to jump over deleted elements. In turn, the interface to the compiler was modified to accommodate a range of new optimization options, adding flags to toggle internal optimizations for each compiled program. We also added a 'fast shutdown mode', enabling users to choose for their program to terminate without freeing memory, and an option to reflect root nodes in a program if desired. Finally, an integration test suite was built for more efficient and sound compiler development.

2.1 Graph Parsing

To resolve the issues with parsing, we decided to employ Judy arrays[1] [44], instead of a simple dynamic array. Invented by Doug Baskins, Judy arrays are a highly cache-optimized hash table implementation. The size of a Judy array is not statically pre-determined and is adjusted, at runtime, to accommodate the number of keys, which themselves can be integers or strings. Instead of storing nodes in the array directly, we also instead store pointers to nodes in the host graph as Judy arrays can only store references to a single word of data. Reallocating the array when doubling it could move the array around and invalidate previous pointers, an issue we resolve in the next section. This allowed an edge to retrieve pointers to its source and target efficiently due to Judy arrays' fast runtime performance [44] [45]. This also resolved problems with unnecessary node array size, allowing node IDs to be arbitrarily large without causing memory problems, as the array simply saw these IDs as meaningless keys in key-value pairs.

[1] `http://judy.sourceforge.net`

2.2 From Arrays to Linked Lists

In order to resolve issues with traversing an array with holes, we decided instead to employ a linked list of nodes, allowing us to jump over any deleted nodes in a single step. We would have new types for entries in the linked list, containing a pointer to the next element and to the current node or edge in question. This would allow us to, for instance, run our discrete graph deletion program in linear time, as the first node in the list could be accessed in constant time.

A performance issue would soon become apparent: should every list element, node and edge not be stored in arrays anymore? Each would have to have a piece of memory allocated for themselves dynamically. Having several calls to `malloc()` for every added node and edge would be highly inefficient, requiring many system calls and memory operations. To resolve this, we decided nodes, edges and linked list entries should all be stored in dynamic arrays, doubling in size when too small.

At this stage, it became apparent that node indices were largely redundant, exposing internal implementation too much and adding unneeded memory operations to resolve a node's address every time it was accessed. Thus, we set about rewriting the runtime to use pointers to nodes and edges rather than indices. This in turn added the problem of pointers being invalidated should the array of nodes or edges be moved when `realloc()` is called to enlarge them. To resolve this problem in turn, we replaced all internal arrays with a new type we dubbed `BigArrays`.

The `BigArray` type, in essence, is an array of pointers to arrays of entries. Each successive array pointed to is double the size of the previous one. The first entry stores two elements, the second four, and so forth. The `BigArray` type is generic, opting to remember the size of its entries and treating each entry as a chunk of memory rather than deal with actual types of entry. Accessing a given index is constant time, using the position of the largest set bit in the index to identify which sub-array to access. Only a logarithmic number of memory allocations are performed overall, with the overall array of arrays being reallocated $O(log(log(n)))$ times - a trivial amount. Moreover, this is an additive value, not multiplicative.

`BigArrays` also contain a static chunk of 160 bytes in themselves, allowing for the first few entries in the array to be stored without having to allocate secondary arrays. When nodes use `BigArrays` of their incident edges, this would mean avoiding unnecessary memory allocations for nodes of small degree, in turn generalizing Bak's solution of storing the first two node indices statically with cleaner control logic.

`BigArrays` also manage holes like the prior implementation did. However, instead of using a second array of holes, `BigArrays` instead store a linked list

of holes within the hole entries in the array, keeping a pointer to the first hole. When a hole is created in the array, that position in the array is overwritten with the data of a new linked list entry, becoming the head of the list of holes. This avoids having to use extra memory for holes, making `BigArrays` more memory efficient and making deletion of elements constant time.

Most importantly, `BigArrays` allow one to allocate more memory to the array without having to possibly move previous entries in memory, simply creating a new array. This means the low number of memory allocations may be maintained without pointers to nodes and edges being invalidated.

Thus, three `BigArrays` are now stored within a graph, one for nodes, one for edges, and one for entries in the linked list of nodes, termed `NodeLists`. A `NodeList` simply contains a pointer to the node it refers to and a pointer to the next entry in the linked list. The same is true of `EdgeLists`, albeit for edges.

Each node now contains a `BigArray` of linked list entries for edges and pointers to the linked list of outgoing edges and of incoming edges. No iteration through edges directly is ever needed beyond printing a graph, which can be done by iterating through the outgoing edges of every node, so no total list of edges is maintained.

A new issue now presented itself: should a node or edge be deleted, all pointers now represent garbage, the element having possibly been overwritten by a hole. This shed light on yet another possible optimization: nodes and edges should remember who references them, and be garbage collected when they are referenced by no one. Nodes now retain flags representing if they are in a graph or referred to in the stack of graph changes, and edges remember if they are in a node's list of incoming/outgoing edges or in the stack also. Should a node or edge ever be deleted, the operation can be deferred should other references still exist. Now, stacks of graph changes can simply store pointers to nodes and edges rather than any data in them, stopping them from being truly deleted but simply ignored by the graph that 'deleted' them, with no linked list entry pointing to them. This garbage collection saves on memory in graph stacks and reduces memory operations.

In light of these modifications, it became apparent that graph copying would be somewhat redundant, as stored pointers would resolve to the original graph and not the copied one. We elected to remove graph copying rather than augment copying because of this, only supporting undoing changes now. Also, all flags for nodes and edges were condensed into a single `char` rather than separate booleans, to save on memory.

2.3 Fast Shutdown Mode

It is good practice, in order to aid runtime of analysis of programs for heap memory bugs, for programs to track and cleanup their allocated heap memory. In particular, when doing this, one must accept a one-time cost when cleaning up the data structures after writing the output graph, during the 'shutdown' step. In order to mitigate this cost, we have introduced a way to turn this off, so that the generated code will simply exit the process as soon as printing the graph. The so called 'fast shutdown' mode can be enabled with a flag passed to the compiler, as documented in Appendix A.

In addition to fast shutdown mode, we have also introduced a more aggressive optimization designed to improve the performance of reduction programs, that will turn off garbage collection of nodes and edges all together, as well as disabling refcounting of host label lists, making garbage collection of host label lists impossible. Once again, documentation of how to enable this more aggressive optimisation is provided in Appendix A.

2.4 Root Reflecting Mode

GP 2 theoretical foundation is rooted graph transformation with relabelling. GP 2 rule schemata are used to instantiate genuine 'rules' which are then applied in the standard way. Unfortunately, Plump and Bak's model results in derivations not being invertible, even when ignoring application conditions. This is because the right square of a derivation need not be a natural pushout. This has the unfortunate consequence that derivations are not invertible. Moreover, that non-roots can be matched against roots, so a rule that was intended to introduce a new root node, might actually behave like `skip`.

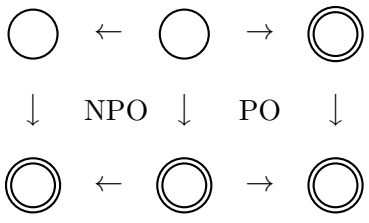


Figure 2.1: Example Rooted Derivation

Campbell has proposed a new foundation for rooted GT systems with relabelling [46] [47], that mitigates this problem. Instead of only insisting on matches preserving root nodes, we also insist on them reflecting them too. This was formalised by defining rootedness using a partial function into a two-point set rather than pointing graphs with root nodes, thus allowing both squares in a derivation to be natural pushouts, where rules are allowed to have

undefined rootedness in their interface graphs.

In order to simulate this new model of rootedness, one only needs to make small modifications to the compiler to enforce reflection of rootedness nodes in matches. We have thus made this change on both the legacy and master branches of the compiler implementation. For usage details, see Appendix A.

2.5 Link-Time Optimisation

In order to produce faster compiled programs, we have increased GCC's optimisation flag from `-O2` to `-O3`[2]. Moreover, we no longer compile the 'library' files ahead of time for linking. We compile them with the generated program, with 'link-time optimisation'[3] enabled, allowing GCC to more aggressively optimise the whole program.

This has lead to a minor change to the CLI interface of the compiler, and also the generated files. Usage documentation can be found in Appendix A.

2.6 Integration Tests

In order to have confidence in the correctness of both the `legacy` and `master` GP 2 Compiler, we have written various integration tests. As of 20th September 2019, there are 140 test cases that are checked. Details of the tests can be found in Appendix B.

[2] `https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html`
[3] `https://gcc.gnu.org/wiki/LinkTimeOptimization`

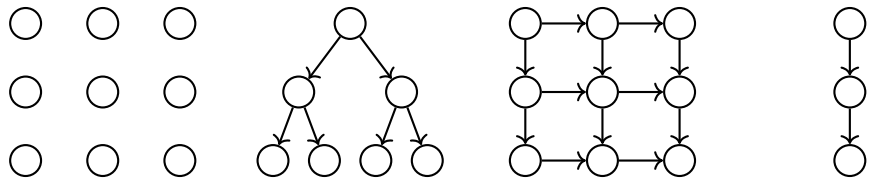
Chapter 3

Timing Results

We conjecture that the new compiler implementation has worst case time complexity no worse than the original implementation. We provide runtime performances of various programs and graph classes as empirical evidence for our conjecture, where timing results differ only by a constant factor. We look at the performance of generation programs, reduction programs, and a couple of other programs, including undirected DFS. We show that there are significant improvements for some types of reduction programs.

Details of the benchmarking software and graph generation can be found in Appendix C, including which compiler flags were used.

3.1 Reduction Performance



(a) Discrete Graph (b) Binary Tree (c) Grid Graph (d) Linked List

Figure 3.1: Input Graph Classes

In Section 1.4, we observed that even the program that simply deleted all isolated nodes could not be executed in linear time by the old implementation. Somewhat more subtle, is when an input graph is connected, however the program splits up the graph as it executes. It is possible to recognise binary DAGs using a slick reduction program (Figure 3.2) with this property. Once again, we observe that the original compiler produces a program that runs in quadratic time on many graph classes, including full binary trees and grid graphs (Figure 3.1). Our new compiler runs in linear time on such graphs.

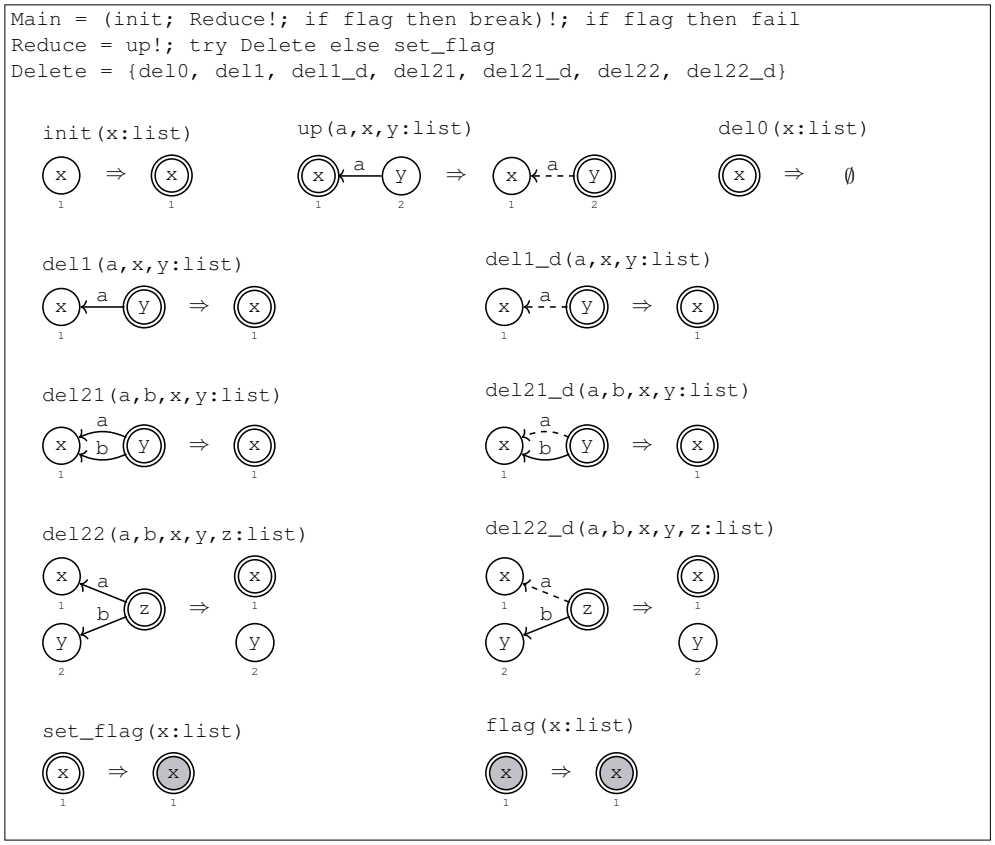
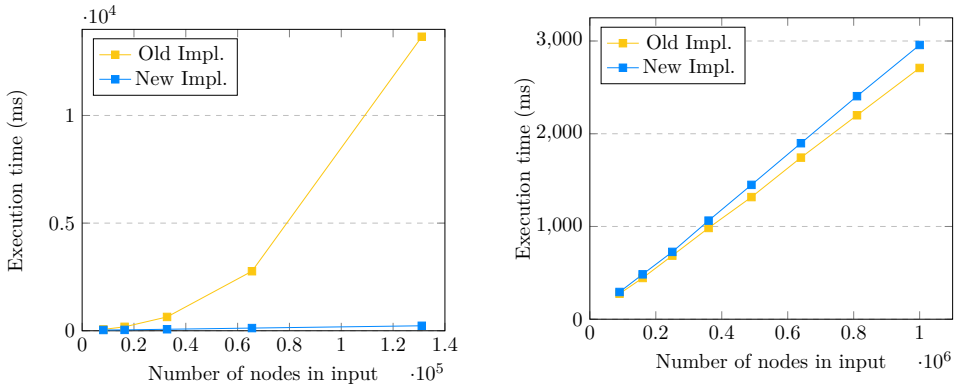


Figure 3.2: GP 2 Program `is-bin-dag.gp2`



(a) Tree Reduction

(b) Grid Reduction

Figure 3.3: Measured Performance of `is-bin-dag.gp2`

Next, we look at a rooted tree reduction program by Campbell [46] (modified the program to work with unmarked graphs) that was linear time on graphs of bounded degree in the previous implementation of the compiler. We confirm that it remains linear time in the new compiler, and include grid graphs as a negative case for the program to determine they are not trees.

```
Main = init; Reduce!; Unmark; if Check then fail
Reduce = {prune0, prune1, push}
Unmark = try unmark; try unmark
Check = {two_nodes, has_loop}
```

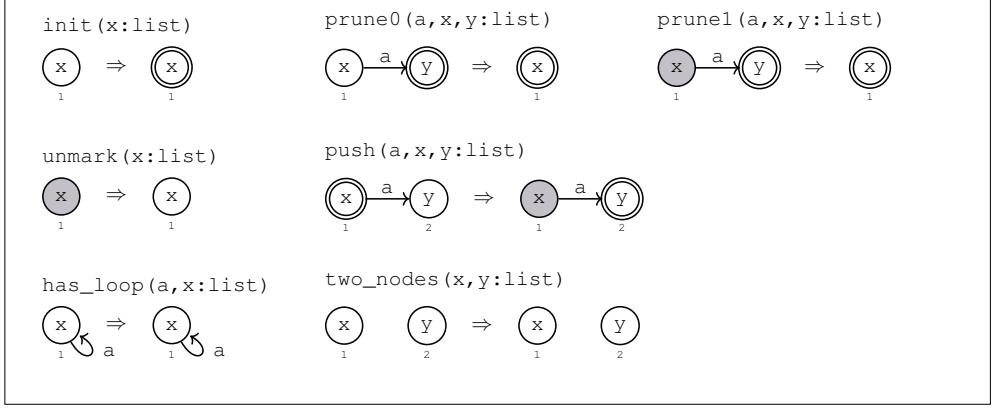
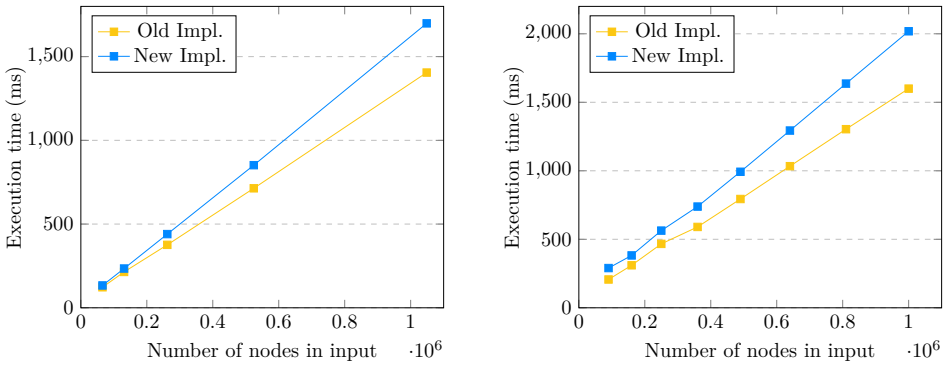


Figure 3.4: GP 2 Program `is-tree.gp2`



(a) Tree Reduction (b) Grid Reduction

Figure 3.5: Measured Performance of `is-tree.gp2`

Finally, we look at recognition of Series-Parallel graphs [48] [22] [23]. We don't expect this program to run in linear time on graphs of bounded degree, but we use this as another example, comparing the runtime performance of the original and new compiler implementations. We include grids as an example of an input that is not Series-Parallel.

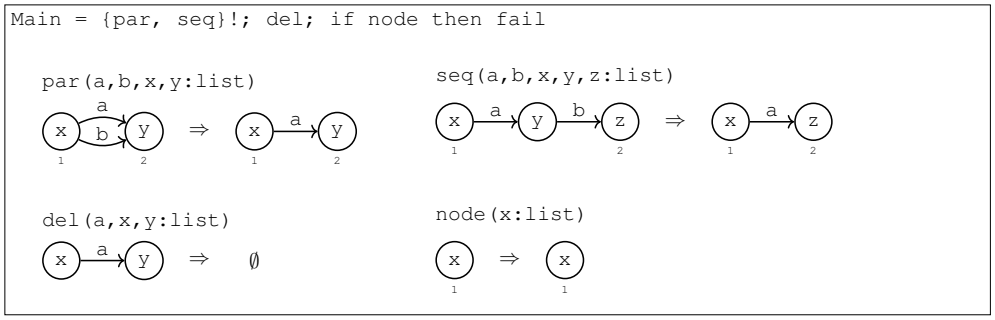
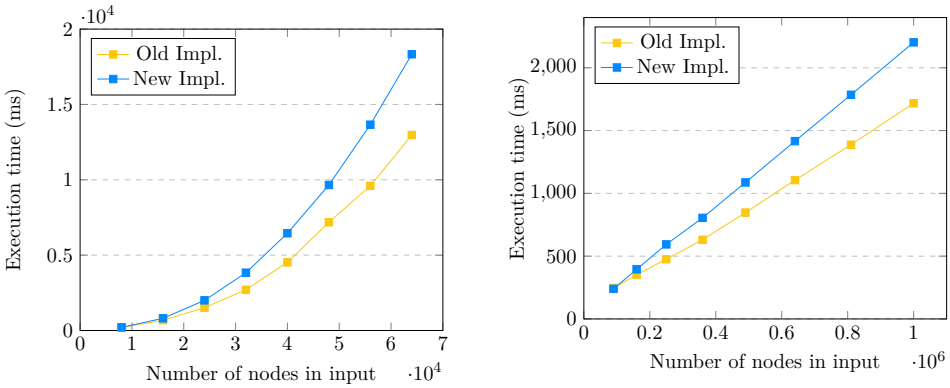


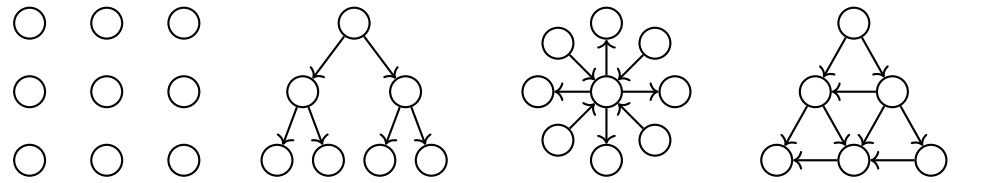
Figure 3.6: GP 2 Program `is-series-par.gp2`



(a) List Reduction (b) Grid Reduction

Figure 3.7: Measured Performance of `is-series-par.gp2`

3.2 Generation Performance



(a) Discrete Graph (b) Binary Tree (c) Star Graph (d) Sierpinski Graph

Figure 3.8: Generated Graph Classes

We don't expect any complexity improvement for 'generation' programs, however we include this class of programs to allow us to verify that this is the case. Our first test case is the generation of discrete graphs, our second is full binary trees, our third is 'star graphs', and our final is Sierpinski graphs using Plump's program, originally written for GP 1 [49].

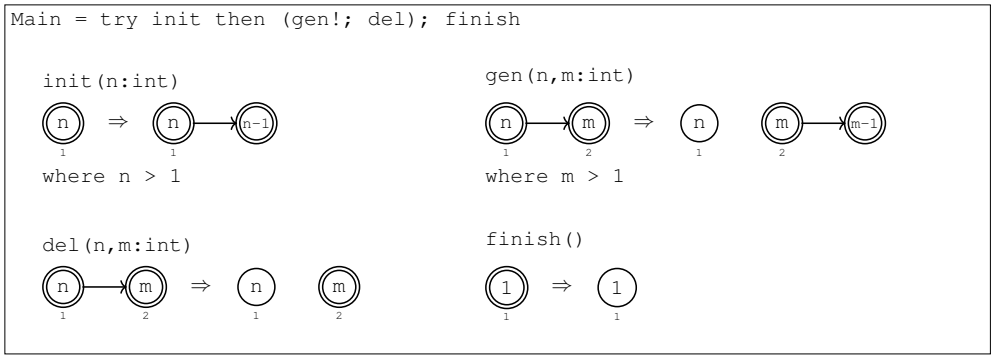


Figure 3.9: GP 2 Program `gen-discrete.gp2`

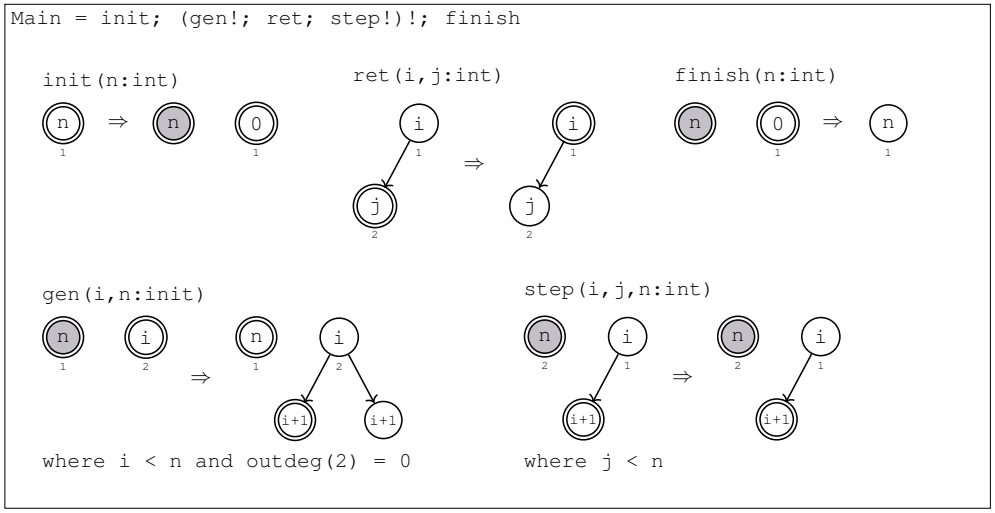


Figure 3.10: GP 2 Program `gen-tree.gp2`

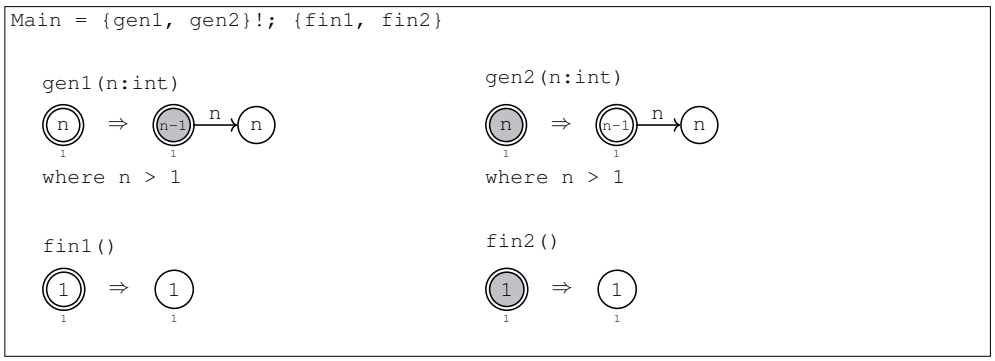


Figure 3.11: GP 2 Program `gen-star.gp2`

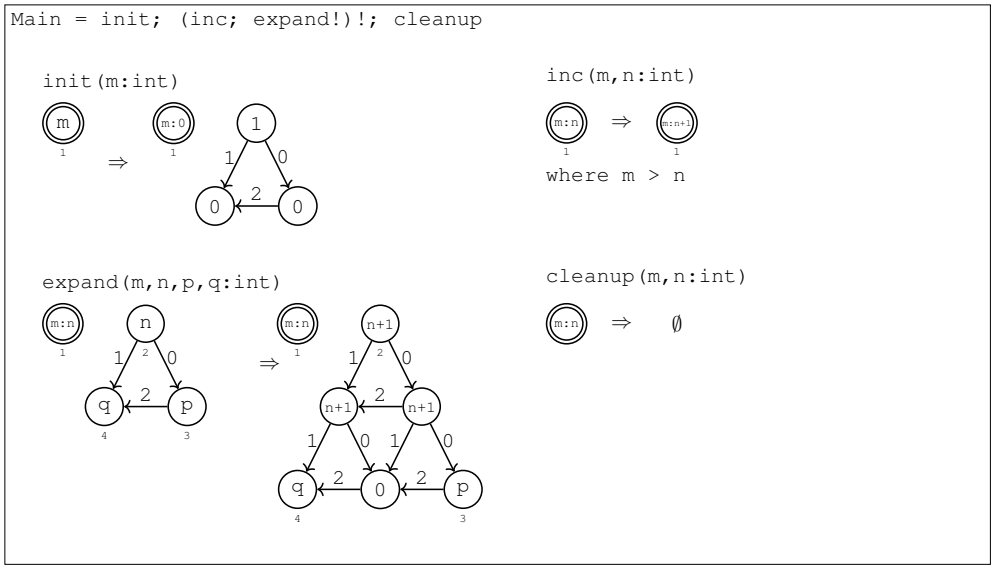
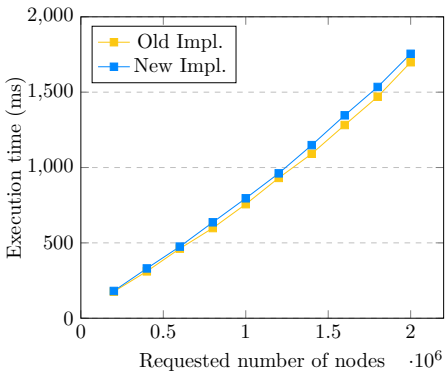
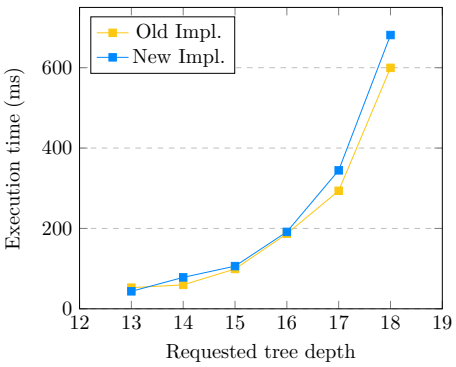


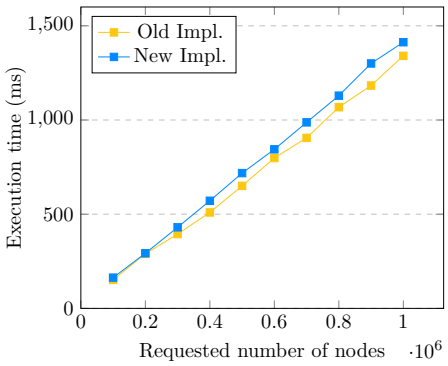
Figure 3.12: GP 2 Program `gen-sierpinski.gp2`



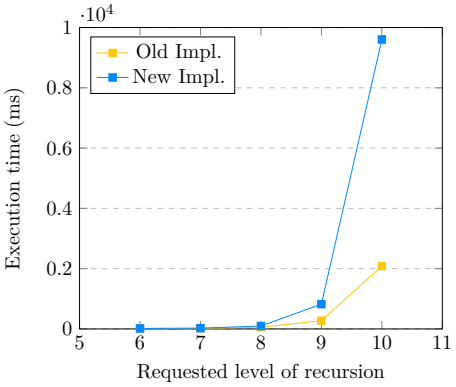
(a) `gen-discrete.gp2` Performance



(b) `gen-tree.gp2` Performance



(c) `gen-star.gp2` Performance



(d) `gen-sierpinski.gp2` Performance

Figure 3.13: Measured Generation Performance

3.3 Other Program Performance

We now look at the performance of undirected DFS [26] by looking at a connected graph recognition program, and also the performance of a transitive closure program. We expect the performance of both the original and new implementations to be similar for connectedness checking, with the new implementation having the edge on grid graphs due to the new implementation of edge lists. In particular, we expect to see good a runtime speedup with the transitive closure program, which is edge search intensive.

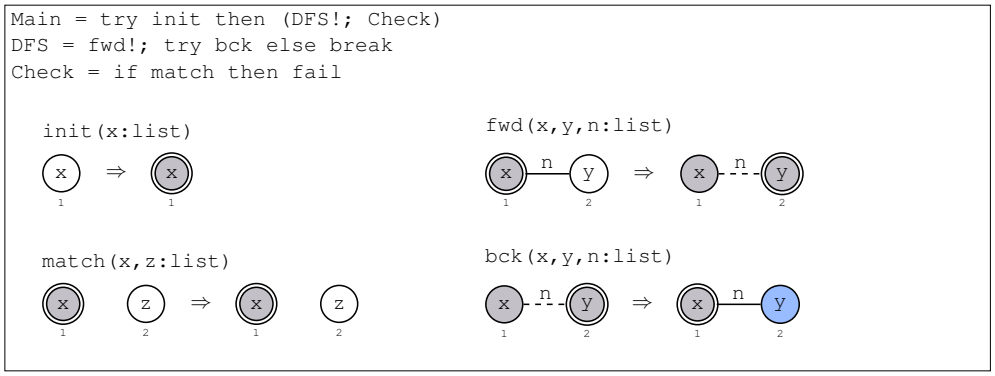
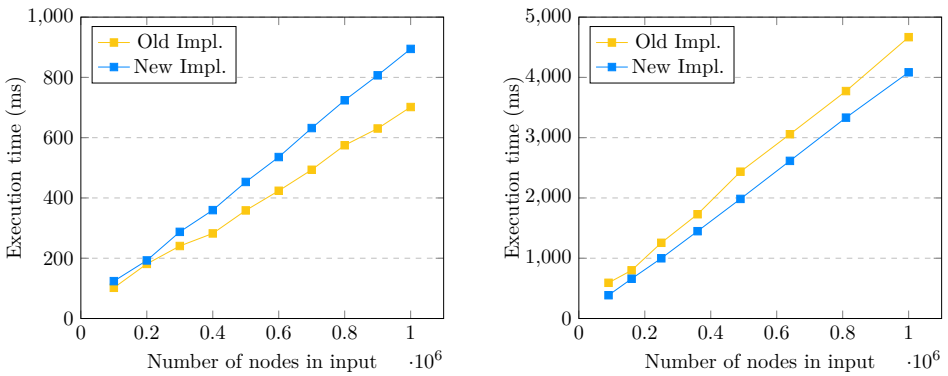


Figure 3.14: GP 2 Program `is-con.gp2`



(a) Performance on Discrete Graphs (b) Performance on Grid Graphs

Figure 3.15: Measured Performance of `is-con.gp2`

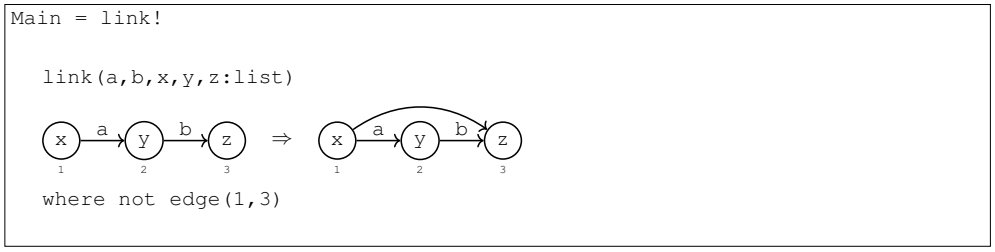
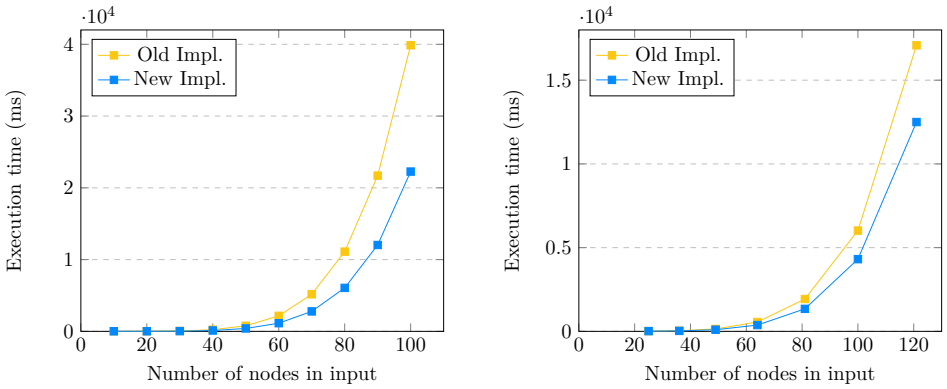


Figure 3.16: GP 2 Program `trans-closure.gp2`



(a) Performance on Linked Lists (b) Performance on Grid Graphs

Figure 3.17: Measured Performance of `trans-closure.gp2`

3.4 Results Summary

To begin, it is evident that the new compiler vastly outperforms the old when executing the `is-discrete.gp2` program, almost appearing to be constant time in comparison. We have clearly reached linear complexity, a feat due to our node list's capacity to skip holes in the underlying node array. The new compiler also outperforms the old when running `trans-closure.gp2` by a significant constant factor. We attribute this to better cache usage; in the legacy compiler, only the indices of the first two edges are statically stored in the `Node` type, whereas the new compiler stores several more direct pointers to nodes statically in a node's `BigArray`. This not only means the locations of more edges are loaded into the cache in one cacheline refill, but also the loaded locations can be resolved in one memory operation rather than two, namely checking the edge array at the given index. We consider this evidence that the new compiler is superior at loading neighboring edges efficiently.

However, the `master` compiler falls short on some test cases; the programs `gen-tree.gp2`, `gen-discrete.gp2`, `gen-star.gp2`, `gen-sierpinski-i.gp2`, `is-tree.gp2` and `is-series-par.gp2` run with a worse constant time factor and the same complexity. Many of these programs only differ by a nearly negligible factor though; the only programs that now perform substantially worse are `gen-sierpinski.gp2`, `is-series-par.gp2` and `is-tree.gp2`. It is possible that `gen-sierpinski.gp2` now performs far worse due to the fact that nodes are now accessed in the reverse order they would have been in the `legacy` compiler, making the failure somewhat artificial. Many of these failings however are likely due to excessive memory operations in the new naive linked list format when iterating through nodes.

Some programs performed better or worse depending on the type of input graph. `is-bin-dag.gp2` saw a boost in performance on linked lists, but a drop on grid graphs. On the contrary, `is-con.gp2` was found to be more

performant on grid graphs but less so on linked lists. This is no consistent drop in performance, but merely a reminder that the internals of the compiler are simply different than before rather than worse.

Overall, the worst case drop in performance was by a constant factor, meaning no complexities were worsened in the observed test cases. Most programs performed similarly to before with a slightly enlarged constant, a byproduct of the memory operations a naive linked list entails.

Chapter 4

Conclusion

A large number of changes were implemented in the new compiler to improve upon the old version. First and foremost, node and edge lists were added to improve iteration performance for programs with a high rate of deletion, letting generated node matching code skip large regions of deleted nodes in constant time.

On top of this, input graph parsing was fixed for graphs of arbitrary size by using Judy arrays. In the previous implementation, a user had to specify the maximum number of nodes and edges expected in input host graphs at program compile time, with the final generated code always allocating this amount of memory regardless of the actual size of input graph. In our updated implementation, this compile time knowledge is obsolete; we now dynamically grow the memory needed during parsing, allowing for arbitrary input graph size with efficient memory consumption.

Also, a method was introduced to shut down the generated program as soon as it has written the output graph without freeing any allocated heap memory properly. In general, proper management of the heap memory is advised to avoid bugs due to memory leaks or otherwise. However, one may wish to avoid doing this in the interest of runtime performance, allowing the process to terminate as quickly as possible. The newly introduced option gives the user this choice.

Finally, an option was added to require matches to both reflect and preserve root nodes. By default, GP 2 will allow a non-root in a rule LHS to match against either a non-root or a root in a host graph, but if we insist on matches reflecting root nodes, matches can only match non-roots against non-roots. With this option employed, rooted rules are now completely reversible and finer control is gained over rule applications.

4.1 Evaluation

Overall, the new `master` compiler has been shown to maintain previous time complexities and improve on the issue of holes in some reduction programs. The `is-discrete.gp2` program has been brought down to linear complexity, making the new compiler more accurately reflect how complexity of graph programs should realistically behave. The new compiler also resolves fundamental issues with the old, such as allowing for arbitrary sizes of input graphs to programs; this makes the compiler fundamentally more correct than it was before.

However, a number of programs performed worse by a constant factor due to the overhead of pointer operations when iterating through a linked list. These issues can be avoided by not using linked lists and iterating through arrays as before, leaving it up to the user to decide how best to compile their program. In any case, no time complexity was worsened. Therefore, we deem this project a success.

4.2 Future Work

It remains future work to determine if it would make sense to add additional node lists for marked vs unmarked to allow a node search to find a marked or unmarked node in constant time. Currently, if one has a graph with arbitrarily many connected components of arbitrary size, visiting all the nodes in the graph with an undirected DFS requires quadratic time. If the DFS were to mark all the nodes in the current component, one could then imagine being able to jump to the next unvisited component in constant time, and then repeating until there are no more components. Obviously, maintaining these additional data structures requires additional time and space, and in general, will not improve the performance of programs that don't need to traverse a graph with arbitrarily many connected components.

As well as this, investigating worsened performance of programs due to linked list operations is an issue to resolve. An option is to have each linked list entry represent a range of node indices in the underlying node array rather than a single node; when iterating through nodes, a list entry may retrieve the array and iterate through it directly within its own range of nodes. When a node is deleted, the linked list entry containing it then fragments into two entries, one for the range of nodes before it and one after. This may help mitigate pointer operations and regain the speed of array iteration while still jumping over holes in the array.

It also remains future work to update the compiler implementation to support integers of arbitrary size. The current implementation supports signed 32-bit integers only, and overflows are not detected during label or condition

evaluation. Similarly, it remains future work to define what happens when one attempts to divide by zero. Currently, division by zero is possible when computing label expression values in rule RHSs and also when evaluating rule conditions. It is possible that one could simply disallow the division operator in conditions, and implicitly have a condition inserted that ensures that division by zero never happens when evaluating RHSs, though its not obvious if this is a reasonable restriction, or if it is better to fail loudly. Another solution could be to introduce a new special runtime error state into the semantics of the language, and insist that any division by zero that occurs to runtime must transition straight to this state and terminate.

There remains an issue with edge conditions appearing negated in boolean expressions, and other complex expressions. This issue is documented on the GitHub issue tracker[1], and affects both the `legacy` and `master` versions of the GP 2 Compiler. For example, the following will result in the compiler producing incorrect code: `not (edge(n0, n1) and edge(n1, n0))`. We are also aware of an issue with `try` statements within the guard of an `if`[2].

Finally, ongoing current research is exploring what classes of graph algorithms can be implemented in linear time in GP 2 using the current compiler. Bak showed (undirected) DFS of graphs of bounded degree and bounded components can be performed in linear time [35] [26], and also 2-colouring of such graphs [50]. Due to this report, we know that we can sometimes perform reduction algorithms that don't limit the growth of the number of graph components in linear time, and Campbell, Courtehoute and Plump recently showed that GP 2 can recognise trees and topologically sort DAGs of bounded degree in linear time [39].

[1] `https://github.com/UoYCS-plasma/GP2/issues/10`
[2] `https://github.com/UoYCS-plasma/GP2/issues/28`

Appendix A

Usage Documentation

In this Appendix, we document the usage of the `legacy` and `master` branches of the GP 2 Compiler, as they stand at 20th September 2019. We also take note of the unofficial Dockerized version of GP2, 'GP2I'[1], and the supporting ecosystem. At time of writing, the GP2 Editor[2] has not been updated to use the `master` version of the compiler, and should be used with the `legacy` branch.

A.1 Legacy Compiler

The `legacy` branch specifies the state of the compiler before our modifications. We keep it available so users may see for themselves the differences in the prior and new compiler versions.

Installing the GP 2 legacy compiler may be done with the following sequence of commands on any standard Linux or Mac computer. If any programs needed are not installed, install them.

```
git clone -b legacy https://github.com/UoYCS-plasma/GP2.git
cd GP2/Compiler
./configure
make
sudo make install
```

A number of flags are available for the GP 2 legacy compiler. In order to validate a given program, a user should write

[1] `https://gitlab.com/YorkCS/Batman/GP2I`
[2] `https://github.com/UoYCS-plasma/GP2-editor`

```
gp2 -p <program_file>
```

where `<program_file>` is the path of the program to be verified. To validate a rule on its own, the rule should be written in its own file and the command

```
gp2 -r <rule_file>
```

should be executed. To validate a graph, the command

```
gp2 -h <graph_file>
```

should be used. Finally, if one wishes to compile a given program, the user should enter the command

```
gp2 [-c] [-d] [-m] [-l <rootdir>] [-o <outdir>]
    [--max-nodes <MAX>] [--max-edges <MAX>] <program_file>
```

A number of options are available, as can be seen. These are as follows:

- `-c`: Use graph copying instead of undoing on the stack.
- `-d`: Compile with GCC debugging flags, if you should wish to use a debugger on your program.
- `-m`: Compile with root reflecting matches.
- `-l <rootdir>`: Specify the root directory for the installed compiler.
- `-o <outdir>`: Specify the path where the compiled files should be placed.
- `--max-nodes <MAX>`: Specify the maximum number of nodes the program's graph parser can handle.
- `--max-edges <MAX>`: Specify the maximum number of edges the program's graph parser can handle.

The latter two options are necessary due to issues noted in the graph parser of the legacy compiler.

The compiler will, when given the necessary program file in this manner, produce a number of C source and header files, along with a Makefile at the specified output directory. When this is done, the user may simply run `make` in this directory to generate the final executable, called `gp2run`. To run this program on a graph, simply enter the `/tmp/gp2` directory and execute

```
./gp2run <graphdir>
```

The output graph will be written in the same path. To delete all the generated files, simply run `make clean`; ensure nothing else is in the current path when doing this.

A.2 New Compiler

The new compiler (on the `master` branch) may be installed by the following means:

```
git clone https://github.com/UoYCS-plasma/GP2.git
cd GP2/Compiler
./configure
make
sudo make install
```

Usage is identical to the legacy version of the compiler. However, some flags have changed:

- `-c`, `--max-nodes`, and `--max-edges` have been removed.
- `-f`: Compile a program in fast shutdown mode, to avoid freeing memory at termination.
- `-g`: Compile a program with minimal garbage collection. This requires fast shutdown to be enabled.
- `-n`: Compile a program without graph node lists, using arrays instead.
- `-q`: Compile a program quickly, without optimizations.
- `-l <libdir>`: Specify the directory for the **library** code.

Also, instead of running `make` to compile the generated C code, one must now run `./build.sh`.

A.3 Dockerized Version

The purpose of the GP2I is to provide a simple Dockerized interface that can be run on almost any AMD64 architecture Linux or MacOS, without needing to install or configure any software, other than Docker. The interface is simple, and abstracts away from the fact there is actually a compiler. One simply specifies the input program and host graph, and GP2I will compile the program using GCC 9.2, and execute it on the host graph. The `legacy` tagged image corresponds to the `legacy` branch of the GP 2 Compiler, and

the `latest` tagged image corresponds to the `master` branch of the GP 2 Compiler.

Running GP2I is simple. Simply choose the `tag`, and relative location of the GP 2 program `prog` and input host graph `host`, as shown below.

```
docker run -v ${PWD}:/data \
  registry.gitlab.com/yorkcs/batman/gp2i:<tag> \
  <prog> <host>
```

It is also possible to specify compiler flags by setting the `GP2_FLAGS` environment variable.

1. On success, the output graph is written to `stdout` and the process exits with code `0`.
2. On compilation error, such as due to an invalid program, the details are written to `stderr`, and the process exits with code `1`.
3. On program error, such as an invalid host graph or program evaluating to `fail`, the details are written to `stderr`, and the process exits with code `2`.

There also exists 'GP2I as a service'. That is, we have a gRPC[3] interface and server implementation that allows client implementation to execute GP 2 programs on host graphs, and receive the program execution time back, along with the output graph or an error message. Originally developed in Summer 2018, there have been minor modifications since, not least, to support the new GP 2 Compiler. The current interface is provided in Figure A.1.

```
1  syntax = "proto3";
2
3  message Request {
4    string program = 1;
5    string graph = 2;
6  }
7
8  message Response {
9    oneof payload {
10     string graph = 1; // the output graph
11     string error = 2; // an error message
12   }
13   uint32 time = 3; // execution time
14 }
15
16 service GP2I {
17   rpc Interpret (Request) returns (Response);
18 }
```

Figure A.1: The `gp2i.proto` Interface Description

[3] `https://grpc.io/`

Appendix B

Software Testing

In this Appendix, we give an account of the integration tests[1] used to give some confidence in the correctness of both the `legacy` and `master` GP 2 Compiler. As of 20th September 2019, there are 140 test cases that are checked, divided into two test suites, `quick` and `slow`. The quick suite contains 138 quick running tests, designed to test the correctness of the compiler on both small, simple test cases, and small, edge cases. The slow suite contains 2 tests that are executed on massive input graphs, designed to check the compiler does not fall over when given very large input graphs.

The tests execute using the GP2I docker images, as described in Section A.3. Root reflecting mode (Section 2.4) is used to test the output graphs are correct up to isomorphism by means of the generated program in Figure B.1, where `$GRAPH` is set to the expected graph.

We have also setup special debug GP2I images that start the generated program using Valgrind[2] to check for memory leaks and other memory errors, such as control flow that depends on uninitialised memory, on the `master` version of GP2. Moreover, we run the quick tests using various different combinations of the compiler flags, although we don't run all the combinations through Valgrind, since fast shutdown mode (Section 2.3) intentionally does not free all allocated heap memory.

[1] `https://gitlab.com/YorkCS/Batman/Tests`
[2] `http://valgrind.org/`

```
1  Main = remove; if {r1, r2, r3, r4} then fail
2
3  remove()
4  $GRAPH
5  =>
6  [ | ]
7  interface={}
8
9  r1(x: list)
10 [ (n1, x) | ]
11 =>
12 [ (n1, x) | ]
13 interface={n1}
14
15 r2(x: list)
16 [ (n1(R), x) | ]
17 =>
18 [ (n1(R), x) | ]
19 interface={n1}
20
21 r3(x: list)
22 [ (n1, x # any) | ]
23 =>
24 [ (n1, x # any) | ]
25 interface={n1}
26
27 r4(x: list)
28 [ (n1(R), x # any) | ]
29 =>
30 [ (n1(R), x # any) | ]
31 interface={n1}
```

Figure B.1: The `iso.gp2` Pseudo-Program

Appendix C

Benchmarking Details

In this Appendix, we give a quick overview of the benchmarking software used in this report. Note that all benchmarks were run on a MacBook Pro (Retina, 15-inch, Mid 2015) with 2.5 GHz Intel Core i7 and 16GB RAM. We support both the new and legacy compilers. For our benchmarking, we set both the max nodes and max edges parameter to 8388608, and when benchmarking the new compiler, we enabled 'fast shutdown mode'.

The benchmarking software uses gRPC, and in particular, the GP2I server described in Section A.3. There are two additional gRPC services. A graph generation service, and a benchmarking service. The graph generation service communicates with the GP2I service in order to generate graphs using GP 2 programs, and the benchmarking service communicates with the graph generation service to generate input graphs, and then with the GP2I service in order to time the execution of programs on input graphs.

There is a benchmarking CLI client that calls the benchmarking service with input programs, and graph generation parameters. The service then streams back progress information for the client to display, and then, once benchmarking is complete, streams the results for the client to display.

```
1  syntax = "proto3";
2
3  message Range {
4    uint32 step = 1;
5    uint32 min = 2;
6    uint32 max = 3;
7  }
8
9  message Config {
10   string mode = 1;
11   Range range = 2;
12 }
```

Figure C.1: The `config.proto` Interface Description

```
syntax = "proto3";

import "config.proto";

message Graph {
  string graph = 1;
  uint32 nodes = 2;
  uint32 edges = 3;
}

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

Figure C.2: The `gen.proto` Interface Description

```
syntax = "proto3";

import "config.proto";

message Program {
  string name = 1;
  string content = 2;
}

message Setup {
  repeated Program programs = 1;
  Config config = 2;
  uint32 runs = 3;
}

message Progress {
  uint32 phase = 1;
  uint32 total = 2;
  uint32 complete = 3;
}

message Benchmark {
  uint32 nodes = 1;
  uint32 edges = 2;
  float rate = 3;
  float time = 4;
}

message Benchmarks {
  string program = 1;
  repeated Benchmark results = 2;
}

message BenchStep {
  oneof payload {
    Progress progress = 1;
    Benchmarks benchmarks = 2;
    string error = 3;
  }
}

service Bench {
  rpc Execute (Setup) returns (stream BenchStep);
}
```

Figure C.3: The `bench.proto` Interface Description

Bibliography

[1] J. Engelfriet and G. Rozenberg, 'Node replacement graph grammars', in *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 1: Foundations*, G. Rozenberg, Ed. World Scientific, 1997, pp. 1–94. DOI: 10.1142/9789812384720_0001.

[2] F. Drewes, H.-J. Kreowski and A. Habel, 'Hyperedge replacement graph grammars', in *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 1: Foundations*, G. Rozenberg, Ed. World Scientific, 1997, pp. 95–162. DOI: 10.1142/9789812384720_0002.

[3] A. Corradini, U. Montanari, F. Rossi, H. Ehrig, R. Heckel and M. Löwe, 'Algebraic approaches to graph transformation. Part I: Basic concepts and double pushout approach', in *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 1: Foundations*, G. Rozenberg, Ed. World Scientific, 1997, pp. 163–245. DOI: 10.1142/9789812384720_0003.

[4] H. Ehrig, R. Heckel, M. Korff, M. Löwe, L. Ribeiro, A. Wagner and A. Corradini, 'Algebraic approaches to graph transformation. Part II: Single pushout approach and comparison with double pushout approach', in *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 1: Foundations*, G. Rozenberg, Ed. World Scientific, 1997, pp. 247–312. DOI: 10.1142/9789812384720_0004.

[5] H. Ehrig, M. Pfender and H. Schneider, 'Graph-grammars: An algebraic approach', in *Proc. 14th Annual Symposium on Switching and Automata Theory (SWAT 1973)*, P. Lewis and J. Brzozowski, Eds., IEEE, 1973, pp. 167–180. DOI: 10.1109/SWAT.1973.11.

[6] H. Ehrig, 'Introduction to the algebraic theory of graph grammars (a survey)', in *Proc. International Workshop on Graph-Grammars and Their Application to Computer Science and Biology*, V. Claus, H. Ehrig and G. Rozenberg, Eds., ser. Lecture Notes in Computer Science, vol. 73, Springer, 1979, pp. 1–69. DOI: 10.1007/BFb0025714.

[7] H. Ehrig, K. Ehrig, U. Prange and G. Taentzer, *Fundamentals of Algebraic Graph Transformation*, ser. Monographs in Theoretical Computer Science. An EATCS Series. Springer, 2006. DOI: 10.1007/3-540-31188-2.

[8] A. Corradini, T. Heindel, F. Hermann and B. König, 'Sesqui-pushout rewriting', in *Proc. Third International Conference on Graph Transformation (ICGT 2006)*, A. Corradini, H. Ehrig, U. Montanari, L. Ribeiro and G. Rozenberg, Eds., ser. Lecture Notes in Computer Science, vol. 4178, Springer, 2006, pp. 30–45. DOI: 10.1007/11841883_4.

[9] V. Danos, T. Heindel, R. Honorato-Zimmer and S. Stucki, 'Reversible sesqui-pushout rewriting', in *Proc. 7th International Conference on Graph Transformation (ICGT 2014)*, H. Giese and B. König, Eds., ser. Lecture Notes in Computer Science, vol. 8571, Springer, 2014, pp. 161–176. DOI: 10.1007/978-3-319-09108-2_11.

[10] A. Habel, J. Müller and D. Plump, 'Double-pushout graph transformation revisited', *Mathematical Structures in Computer Science*, vol. 11, no. 5, pp. 637–688, 2001. DOI: 10.1017/S0960129501003425.

[11] O. Runge, C. Ermel and G. Taentzer, 'AGG 2.0 – new features for specifying and analyzing algebraic graph transformations', in *Proc. 4th International Symposium on Applications of Graph Transformations with Industrial Relevance (AGTIVE 2011)*, A. Schürr, D. Varró and G. Varró, Eds., ser. Lecture Notes in Computer Science, vol. 7233, Springer, 2011, pp. 81–88. DOI: 10.1007/978-3-642-34176-2_8.

[12] M. Hannachi, I. Rodriguez, K. Drira, H. Pomares and E. Saul, 'GMTE: A tool for graph transformation and exact/inexact graph matching', in *Proc. 9th IAPR-TC-15 International Workshop on Graph-Based Representations in Pattern Recognition (GbRPR 2013)*, W. Kropatsch, N. Artner, Y. Haxhimusa and X. Jiang, Eds., ser. Lecture Notes in Computer Science, vol. 7877, Springer, 2013, pp. 71–80. DOI: 10.1007/978-3-642-38221-5_8.

[13] J. Glauert, J. Kennaway and M. Sleep, 'Dactl: An experimental graph rewriting language', in *Proc. 4th International Workshop on Graph Grammars and Their Application to Computer Science (1990)*, H. Ehrig, H.-J. Kreowski and G. Rozenberg, Eds., ser. Lecture Notes in Computer Science, vol. 532, Springer, 1991, pp. 378–395. DOI: 10.1007/BFb0017401.

[14] D. Plump, 'The design of GP 2', in *Proc. 10th International Workshop on Reduction Strategies in Rewriting and Programming (WRS 2011)*, S. Escobar, Ed., ser. Electronic Proceedings in Theoretical Computer Science, vol. 82, Open Publishing Association, 2012, pp. 1–16. DOI: 10.4204/EPTCS.82.1.

[15] A. Agrawal, G. Karsai, S. Neema, F. Shi and A. Vizhanyo, 'The design of a language for model transformations', *Software & Systems Modeling*, vol. 5, no. 3, pp. 261–288, 2006. DOI: 10.1007/s10270-006-0027-7.

[16] A. Ghamarian, M. de Mol, A. Rensink, E. Zambon and M. Zimakova, 'Modelling and analysis using GROOVE', *International Journal on Software Tools for Technology Transfer*, vol. 14, no. 1, pp. 15–40, 2012. DOI: 10.1007/s10009-011-0186-x.

[17] E. Jakumeit, S. Buchwald and M. Kroll, 'GrGen.NET – the expressive, convenient and fast graph rewrite system', *International Journal on Software Tools for Technology Transfer*, vol. 12, no. 3–4, pp. 263–271, 2010. DOI: 10.1007/s10009-010-0148-8.

[18] T. Arendt, E. Biermann, S. Jurack, C. Krause and G. Taentzer, 'Henshin: Advanced concepts and tools for in-place EMF model transformations', in *Proc. 13th International Conference on Model Driven Engineering Languages and Systems (MODELS 2010)*, D. Petriu, N. Rouquette and Ø. Haugen, Eds., ser. Lecture Notes in Computer Science, vol. 6394, Springer, 2010, pp. 121–135. DOI: 10.1007/978-3-642-16145-2_9.

[19] A. Schürr, A. Winter and A. Zündorf, 'The PROGRES approach: Language and environment', in *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 2: Applications, Languages and Tools*, H. Ehrig, G. Engels, H.-J. Kreowski and G. Rozenberg, Eds. World Scientific, 1999, pp. 551–603. DOI: 10.1142/9789812815149_0014.

[20] M. Fernández, H. Kirchner, I. Mackie and B. Pinaud, 'Visual modelling of complex systems: Towards an abstract machine for PORGY', in *Proc. 10th Conference on Computability in Europe (CiE 2014)*, A. Beckmann, E. Csuhaj-Varjú and K. Meer, Eds., ser. Lecture Notes in Computer Science, vol. 8493, Springer, 2014, pp. 183–193. DOI: 10.1007/978-3-319-08019-2_19.

[21] A. Habel and D. Plump, 'Computational completeness of programming languages based on graph transformation', in *Proc. 4th International Conference on Foundations of Software Science and Computation Structures (FOSSACS 2001)*, F. Honsell and M. Miculan, Eds., ser. Lecture Notes in Computer Science, vol. 2030, Springer, 2001, pp. 230–245. DOI: 10.1007/3-540-45315-6_15.

[22] D. Plump, 'The graph programming language GP', in *Proc. Third International Conference on Algebraic Informatics (CAI 2009)*, S. Bozapalidis and G. Rahonis, Eds., ser. Lecture Notes in Computer Science, vol. 5725, Springer, 2009, pp. 99–122. DOI: 10.1007/978-3-642-03564-7_6.

[23] ——, 'Reasoning about graph programs', in *Proc. 9th International Workshop on Computing with Terms and Graphs (TERMGRAPH 2016)*, A. Corradini and H. Zantema, Eds., ser. Electronic Proceedings in Theoretical Computer Science, vol. 225, Open Publishing Association, 2016, pp. 35–44. DOI: 10.4204/EPTCS.225.6.

[24] A. Habel and D. Plump, 'Relabelling in graph transformation', in *Proc. First International Conference on Graph Transformation (ICGT 2002)*, A. Corradini, H. Ehrig, H.-J. Kreowski and G. Rozenberg, Eds., ser. Lecture Notes in Computer Science, vol. 2505, Springer, 2002, pp. 135–147. DOI: 10.1007/3-540-45832-8_12.

[25] G. Plotkin, 'A structural approach to operational semantics', *The Journal of Logic and Algebraic Programming*, vol. 60–61, pp. 17–139, 2004. DOI: 10.1016/j.jlap.2004.05.001.

[26] C. Bak, 'GP 2: Efficient implementation of a graph programming language', PhD thesis, Department of Computer Science, University of York, UK, 2015. [Online]. Available: https://etheses.whiterose.ac.uk/12586/.

[27] C. Poskitt and D. Plump, 'Hoare-style verification of graph programs', *Fundamenta Informaticae*, vol. 118, no. 1–2, pp. 135–175, 2012. DOI: 10.3233/FI-2012-708.

[28] ——, 'Verifying total correctness of graph programs', in *Selected Revised Papers from the 4th International Workshop on Graph Computation Models (GCM 2012)*, R. Echahed, A. Habel and M. Mosbah, Eds., ser. Electronic Communications of the EASST, vol. 61, 2013. DOI: 10.14279/tuj.eceasst.61.827.

[29] C. Poskitt, 'Verification of graph programs', PhD thesis, Department of Computer Science, University of York, UK, 2013. [Online]. Available: https://etheses.whiterose.ac.uk/4700/.

[30] C. Poskitt and D. Plump, 'Verifying monadic second-order properties of graph programs', in *Proc. 7th International Conference on Graph Transformation (ICGT 2014)*, H. Giese and B. König, Eds., ser. Lecture Notes in Computer Science, vol. 8571, Springer, 2014, pp. 33–48. DOI: 10.1007/978-3-319-09108-2_3.

[31] C. A. R. Hoare, 'An axiomatic basis for computer programming', *Communications of the ACM*, vol. 12, no. 10, pp. 576–580, 1969. DOI: 10.1145/363235.363259.

[32] G. Manning and D. Plump, 'The GP programming system', in *Proc. 7th International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2008)*, C. Ermel, R. Heckel and J. de Lara, Eds., ser. Electronic Communications of the EASST, vol. 10, 2008. DOI: 10.14279/tuj.eceasst.10.150.

[33] I. Hristakiev and D. Plump, 'Checking graph programs for confluence', in *Software Technologies: Applications and Foundations – STAF 2017 Collocated Workshops, Revised Selected Papers*, M. Seidl and S. Zschaler, Eds., ser. Lecture Notes in Computer Science, vol. 10748, Springer, 2018, pp. 92–108. DOI: 10.1007/978-3-319-74730-9_8.

[34] I. Hristakiev, 'Confluence analysis for a graph programming language', PhD thesis, Department of Computer Science, University of York, UK, 2018. [Online]. Available: https://etheses.whiterose.ac.uk/20255/.

[35] C. Bak and D. Plump, 'Rooted graph programs', in *Proc. 7th International Workshop on Graph Based Tools (GraBaTs 2012)*, C. Krause and B. Westfechtel, Eds., ser. Electronic Communications of the EASST, vol. 54, 2012. DOI: 10.14279/tuj.eceasst.54.780.

[36] D. Plump, 'From imperative to rule-based graph programs', *Journal of Logical and Algebraic Methods in Programming*, vol. 88, pp. 154–173, 2017. DOI: 10.1016/j.jlamp.2016.12.001.

[37] T. Atkinson, D. Plump and S. Stepney, 'Evolving graphs by graph programming', in *Proc. 21st European Conference on Genetic Programming (EuroGP 2018)*, M. Castelli, L. Sekanina, M. Zhang, S. Cagnoni and P. García-Sánchez, Eds., ser. Lecture Notes in Computer Science, vol. 10781, Springer, 2018, pp. 35–51. DOI: 10.1007/978-3-319-77553-1_3.

[38] ——, 'Probabilistic graph programs for randomised and evolutionary algorithms', in *Proc. 11th International Conference on Graph Transformation (ICGT 2018)*, L. Lambers and J. Weber, Eds., ser. Lecture Notes in Computer Science, vol. 10887, Springer, 2018, pp. 63–78. DOI: 10.1007/978-3-319-92991-0_5.

[39] G. Campbell, B. Courtehoute and D. Plump, 'Linear-time graph algorithms in GP 2', in *Proc. 8th Conference on Algebra and Coalgebra in Computer Science (CALCO 2019)*, M. Roggenbach and A. Sokolova, Eds., ser. Leibniz International Proceedings in Informatics (LIPIcs), To appear, vol. 139, Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2019.

[40] G. Campbell, J. Romo and D. Plump, 'Fast graph programs', Department of Computer Science, University of York, UK, Tech. Rep., 2018. [Online]. Available: https://cdn.gjcampbell.co.uk/2018/Fast-Graph-Programs.pdf.

[41] H. Dörr, *Efficient Graph Rewriting and its Implementation*, ser. Lecture Notes in Computer Science. Springer, 1995, vol. 922. DOI: 10.1007/BFb0031909.

[42] A. Elliot, 'Towards an integrated development environment for GP 2', MEng Thesis, Department of Computer Science, University of York, UK, 2013.

[43] S. Hand, 'A graphical editor for GP 2', BSc Thesis, Department of Computer Science, University of York, UK, 2019.

[44] A. Silverstein, 'Judy iv shop manual', Hewlett-Packard, Tech. Rep., 2002. [Online]. Available: http://judy.sourceforge.net/application/shop_interm.pdf.

[45] H. Luan, X. Du, S. Wang, Y. Ni and Q. Chen, 'J$^+$-tree: A new index structure in main memory', in *Proc. 12th International Conference on Database Systems for Advanced Applications (DASFAA 2007)*, R. Kotagiri, P. R. Krishna, M. Mohania and E. Nantajeewarawat, Eds., ser. Lecture Notes in Computer Science, vol. 4443, Springer, 2007, pp. 386–397. DOI: 10.1007/978-3-540-71703-4_34.

[46] G. Campbell, 'Efficient graph rewriting', BSc Thesis, Department of Computer Science, University of York, UK, 2019. [Online]. Available: https://cdn.gjcampbell.co.uk/2019/BSc-Project.pdf, Erratum-ibid. Campbell [47].

[47] ——, 'Errata for bsc thesis: Efficient graph rewriting', Department of Computer Science, University of York, UK, Erratum, 2019. [Online]. Available: https://cdn.gjcampbell.co.uk/2019/BSc-Project-Errata.pdf.

[48] R. J. Duffin, 'Topology of series-parallel networks', *Journal of Mathematical Analysis and Applications*, vol. 10, no. 2, pp. 303–318, 1965. DOI: 10.1016/0022-247X(65)90125-3.

[49] G. Taentzer, E. Biermann, D. Bisztray, B. Boneva, A. Boronat, L. Geiger, R. Geiß, Á. Horvath, O. Kniemeyer, T. Mens, B. Ness, D. Plump and T. Vajk, 'Generation of sierpinski triangles: A case study for graph transformation tools', in *Proc. Third International Symposium on Applications of Graph Transformations with Industrial Relevance (AGTIVE 2007)*, A. Schürr, M. Nagl and A. Zündorf, Eds., ser. Lecture Notes in Computer Science, vol. 5088, Springer, 2008, pp. 514–539. DOI: 10.1007/978-3-540-89020-1_35.

[50] C. Bak and D. Plump, 'Compiling graph programs to C', in *Proc. 9th International Conference on Graph Transformation (ICGT 2016)*, R. Echahed and M. Minas, Eds., ser. Lecture Notes in Computer Science, vol. 9761, Springer, 2016, pp. 102–117. DOI: 10.1007/978-3-319-40530-8_7.