The Improved GP 2 Compiler

11th International Workshop on Graph Computation Models

Graham Campbell
School of Mathematics, Statistics and Physics, Newcastle University, UK

Jack Romö
Mathematical Institute, University of Oxford, UK

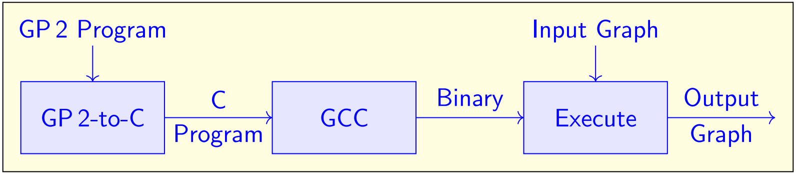
Detlef Plump
Department of Computer Science, University of York, UK

June 2020

Graph Programming Language GP 2

An experimental DSL for graphs, based on attributed DPO graph-transformation rules. GP 2:

- abstracts from low-level data structures;
- has a formal operational semantics (Plump 2012; Bak 2015);
- aims to facilitate formal reasoning on programs (Poskitt and Plump 2012; Plump 2016; Wulandari and Plump 2020);
- is computationally complete (Plump 2017).



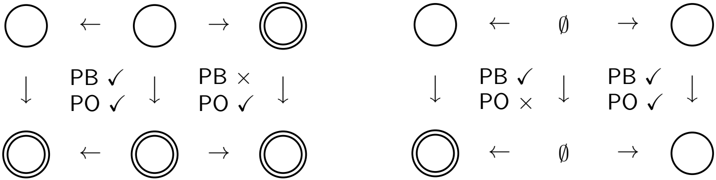
Making GP 2 Fast

Performance bottleneck: matching the left-hand graph L of a rule within a host graph G, requiring time polynomial in the size of L.

- Linear-time graph algorithms in imperative languages may be slowed 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 (Dörr 1995; Bak and Plump 2012).
- Roots in rules must match roots in the host graph so that matches are restricted to the neighbourhood of the host graph's roots.
- Using rooted rules, some graph algorithms can be implemented to run in linear time on graphs of bounded degree: computing a 2-colouring (Bak and Plump 2016); topological sorting of acyclic graphs (Campbell, Courtehoute, and Plump 2019). MSTs can be computed in linearithmic time (Courtehoute and Plump 2020).

Old Theoretical Issue

The theory of rooted graph transformation has some undesirable properties, because morphisms need not be root-reflecting.



(a) Not invertible

(b) Not DPO rewriting

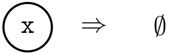
The 2nd example is due to Plump and Wulandari (unpublished). These problems were fixed by Campbell (2019) by requiring root-reflecting morphisms for rooted graph transformation with relabelling.

Old Implementation Issue

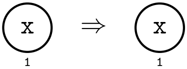
The internal representation of graphs adds a linear factor to the runtime of some programs where one would not expect.

```
Main = del!; if node then fail
```

del(x:list)



node(x:list)



The above program evaluates to `fail` if and only if the input graph is discrete (has no edges). We would expect it to run in linear time, but it actually takes quadratic time on discrete graphs!

Old Compiler: Graph Data Structure

The original compiler stored a graph as a dynamic array of nodes and another of edges.

- Each of these contains two arrays, one of the actual elements and another of indices that are empty, or “holes”.
- When iterating through nodes, each index has to be checked to ensure it is not a hole.
- Deleting a node requires tracking the new hole. Inserting a node can be done by filling a hole should one exist.
- Repeatedly deleting a large number of nodes is expensive.

Root nodes are tracked by a linked list, each entry holding a pointer to a root node. Iterating through or deleting root nodes takes constant time, if there is an upper bound on the number of root nodes.

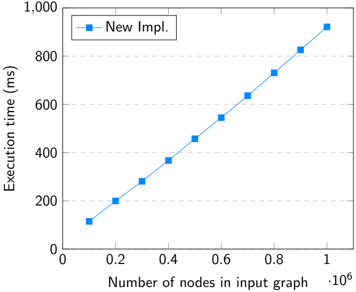
New Compiler: Graph Data Structure

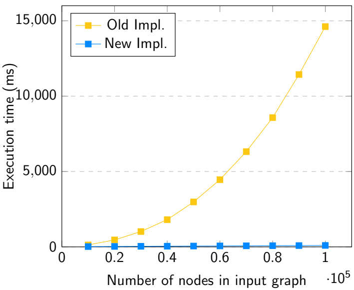
To resolve the problems hole arrays pose, we switched to a linked list pointing to nodes.

- Now the recognition program for discrete graph runs in linear time, as holes are skipped that previously were traversed.
- Other data structures have faster random access time, such as balanced binary trees, but our only use case is iterating through the entire list to match subgraphs or adding/deleting nodes and edges.

There were lots of other internal changes to enable this, such as the replacement of integer IDs by pointers, and re-implementation of edge lists which leads to good performance of our next examples.

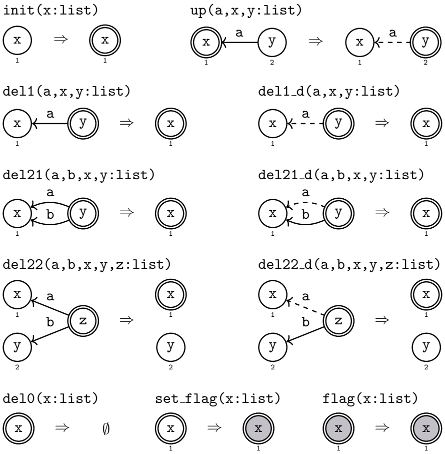
Discrete Graph Testing Timing



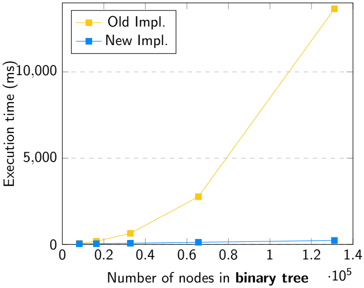


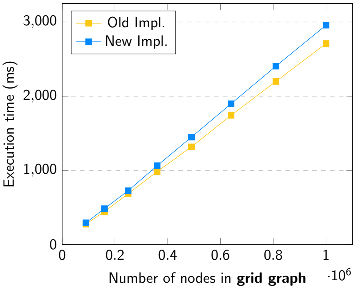
Binary DAG Testing Program

```
Main = (init; Reduce!; if flag then break)!; if flag then fail
Reduce = up!; try Delete else set_flag
Delete = {del1, del1_d, del21, del21_d, del22, del22_d, del0}
```



Binary DAG Testing Timing

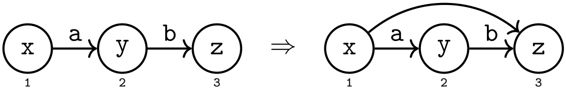




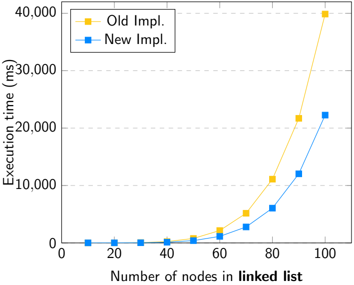
Transitive Closure Program

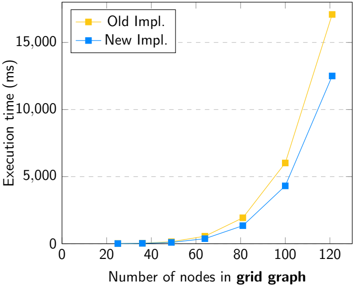
```
Main = link!

   link(a,b,x,y,z:list)
```




```
   where not edge(1,3)
```





Future Work

- Faster matching/detection of nodes with a specific mark.
- Overcoming the bounded degree restriction for constant time matching (partially answered, in progress).
- Which classes of graph algorithms have linear time rule-based implementations (in GP 2)? Can we recognise series-parallel graphs in linear time in a rule-based language?
- A formal model of complexity for GP 2. We don't know of any rule based graph programming language which has one, actually!
- (Semi)automated refinement of GP 2 programs without root nodes, to fast programs with root nodes (hard!).

https://github.com/UoYCS-plasma/GP2

References I

Bak, Christopher (2015). "GP 2: Efficient Implementation of a Graph Programming Language". PhD thesis. Department of Computer Science, University of York, UK. URL: https://etheses.whiterose.ac.uk/12586/.

Bak, Christopher and Detlef Plump (2012). "Rooted Graph Programs". In: *Proc. 7th International Workshop on Graph Based Tools (GraBaTs 2012)*. Ed. by Christian Krause and Bernhard Westfechtel. Vol. 54. Electronic Communications of the EASST. DOI: 10.14279/tuj.eceasst.54.780.

— (2016). "Compiling Graph Programs to C". In: *Proc. 9th International Conference on Graph Transformation (ICGT 2016)*. Ed. by Rachid Echahed and Mark Minas. Vol. 9761. Lecture Notes in Computer Science. Springer, pp. 102–117. DOI: 10.1007/978-3-319-40530-8_7.

Campbell, Graham (2019). "Efficient Graph Rewriting". BSc Thesis. Department of Computer Science, University of York, UK. URL: https://arxiv.org/abs/1906.05170.

Campbell, Graham, Brian Courtehoute, and Detlef Plump (2019). "Linear-Time Graph Algorithms in GP 2". In: *Proc. 8th Conference on Algebra and Coalgebra in Computer Science (CALCO 2019)*. Ed. by Markus Roggenbach and Ana Sokolova. Vol. 139. Leibniz International Proceedings in Informatics (LIPIcs). Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 16:1–16:23. DOI: 10.4230/LIPIcs.CALCO.2019.16.

Courtehoute, Brian and Detlef Plump (2020). "A Fast Graph Program for Computing Minimum Spanning Trees". In: *Pre-Proc. 11th International Workshop on Graph Computation Models (GCM 2020)*. Ed. by Berthold Hoffmann and Mark Minas, pp. 165–183.

Dörr, Heiko (1995). *Efficient Graph Rewriting and its Implementation*. Vol. 922. Lecture Notes in Computer Science. Springer. DOI: 10.1007/BFb0031909.

References II

Plump, Detlef (2012). "The Design of GP 2". In: *Proc. 10th International Workshop on Reduction Strategies in Rewriting and Programming (WRS 2011)*. Ed. by Santiago Escobar. Vol. 82. Electronic Proceedings in Theoretical Computer Science. Open Publishing Association, pp. 1–16. DOI: 10.4204/EPTCS.82.1.

— (2016). "Reasoning about Graph Programs". In: *Proc. 9th International Workshop on Computing with Terms and Graphs (TERMGRAPH 2016)*. Ed. by Andrea Corradini and Hans Zantema. Vol. 225. Electronic Proceedings in Theoretical Computer Science. Open Publishing Association, pp. 35–44. DOI: 10.4204/EPTCS.225.6.

— (2017). "From Imperative to Rule-Based Graph Programs". In: *Journal of Logical and Algebraic Methods in Programming* 88, pp. 154–173. DOI: 10.1016/j.jlamp.2016.12.001.

Poskitt, Christopher and Detlef Plump (2012). "Hoare-Style Verification of Graph Programs". In: *Fundamenta Informaticae* 118.1–2, pp. 135–175. DOI: 10.3233/FI-2012-708.

Wulandari, Gia and Detlef Plump (2020). "Verifying Graph Programs with First-Order Logic". In: *Pre-Proc. 11th International Workshop on Graph Computation Models (GCM 2020)*. Ed. by Berthold Hoffmann and Mark Minas, pp. 184–205.