UNIVERSITY of York

This is a repository copy of *Compiling Graph Programs to C*.

White Rose Research Online URL for this paper:
https://eprints.whiterose.ac.uk/126396/

Version: Accepted Version

Proceedings Paper:
Bak, Christopher Peter and Plump, Detlef orcid.org/0000-0002-1148-822X (2016) Compiling Graph Programs to C. In: Echahed, Rachid and Minas, Mark, (eds.) Proceedings 9th International Conference on Graph Transformation (ICGT 2016). 9th International Conference on Graph Transformation (ICGT 2016), 05-06 Jul 2016 Lecture Notes in Computer Science . Springer , AUT , pp. 102-117.

https://doi.org/10.1007/978-3-319-40530-8_7

Reuse
Items deposited in White Rose Research Online are protected by copyright, with all rights reserved unless indicated otherwise. They may be downloaded and/or printed for private study, or other acts as permitted by national copyright laws. The publisher or other rights holders may allow further reproduction and re-use of the full text version. This is indicated by the licence information on the White Rose Research Online record for the item.

Takedown
If you consider content in White Rose Research Online to be in breach of UK law, please notify us by emailing eprints@whiterose.ac.uk including the URL of the record and the reason for the withdrawal request.

White Rose university consortium
Universities of Leeds, Sheffield & York

eprints@whiterose.ac.uk
https://eprints.whiterose.ac.uk/

Compiling Graph Programs to C

Christopher Bak* and Detlef Plump

The University of York, United Kingdom
(cb574|detlef.plump)@york.ac.uk

Abstract. We show how to generate efficient C code for a high-level domain-specific language for graphs. The experimental language GP 2 is based on graph transformation rules and aims to facilitate formal reasoning on programs. Implementing graph programs is challenging because rule matching is expensive in general. GP 2 addresses this problem by providing *rooted* rules which under mild conditions can be matched in constant time. Using a search plan, our compiler generates C code for matching rooted graph transformation rules. We present run-time experiments with our implementation in a case study on checking graphs for two-colourability: on grid graphs of up to 100,000 nodes, the compiled GP 2 program is as fast as the tailor-made C program given by Sedgewick.

1 Introduction

GP 2 is an experimental domain-specific language for graphs whose basic command is the application of graph transformation rules. The language has a simple syntax and semantics to support formal reasoning on programs (see [14] for a Hoare-logic approach to verifying graph programs). GP 2's initial implementation is an interpreter running in one of two modes, either fully exploring the non-determinism inherent to transformation rules or attempting to produce a single result [3]. In this paper, we report on a compiler for GP 2 which translates programs directly into efficient C code.

The bottleneck for generating fast code for 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 $\text{size}(G)^{\text{size}(L)}$ (which is polynomial since L is fixed). As a consequence, linear-time imperative programs operating on graphs may be slowed down to polynomial time when they are recast as rule-based graph programs. To speed up graph matching, GP 2 allows to distinguish some nodes in rules and host graphs as so-called roots, and to match roots in rules with roots in host graphs. This concept goes back to Dörr [7] and was also studied by Dodds and Plump [6].

Our compiler, described in Section 3, translates GP 2 source code directly into C code, bridging the large gap between graph transformation rules and C. We use a *search plan* to generate code for graph matching, deconstructing each matching step into a sequence of primitive matching operations from which structured code is generated. The code generated to evaluate rule conditions

* This author's work was partially supported by an EPSRC Doctoral Training Grant

is interleaved in the matching code such that conditions are evaluated as soon as their parameters are assigned values, to rule out invalid matches at the first opportunity. Another non-standard aspect of the compiler is that programs are analysed to establish when the state (host graph) needs to be recorded for potential backtracking at runtime. Backtracking is required by GP 2's transaction-like branching constructs `if-then-else` and `try-then-else` which may contain arbitrary subprograms as guards.

In [4] we identified *fast* rules, a large class of conditional rooted rules, and proved that they can be applied in constant time if host graphs have a bounded node degree (an assumption often satisfied in practice). In Section 4, we demonstrate the practicality of rooted graph programs with fast rules in a case study on graph colouring: we give a GP 2 program that 2-colours host graphs in linear time. We show that on grid graphs of up to 100,000 nodes, the compiled GP 2 program matches the speed of Sedgewick's tailor-made implementation in C [17]. In this way, users get the best of both worlds: they can write visual, high-level graph programs with the performance of a relatively low-level language.

2 The Graph Programming Language GP 2

GP 2 is the successor to the graph programming language GP [12]. This section gives a brief introduction to GP 2. The original language definition is [13], an up-to-date version is given in the PhD thesis of the first author [2].

2.1 Conditional Rule Schemata

GP 2's principal programming constructs are conditional rule schemata (abbreviated to *rule schemata* or, when the context is clear, *rules*). Rule schemata extend standard graph transformation rules[1] with expressions in labels and with application conditions. Figure 1 shows the declaration of a conditional rule schema `rule`. The numbered nodes are the *interface* nodes. Nodes that are in the left-hand side but not in the interface are deleted by the rule. Similarly, nodes that are in the right-hand side but not in the interface are added.

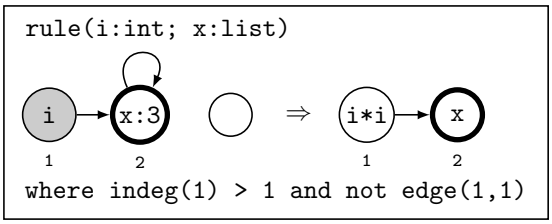


Fig. 1: Declaration of a conditional rule schema

[1] in the double-pushout approach with injective matching

The top line of the declaration states the name of the rule schema and lists the variables that are used in the labels and in the condition. All variables occurring in the right-hand side and in the condition must also occur in the left-hand side because their values a runtime are determined by matching the left-hand side with a subgraph of the host graph.

Each variable is declared with a type which is either `int`, `char`, `string`, `atom` or `list`. Types form a subtype hierarchy in which integers and character strings are basic types, both of which are atoms. Atoms in turn are considered as lists of length one. Labels in host graphs are variable-free expressions containing only constructor operations such as list or string concatenation. Lists are constructed by the colon operator which denotes list concatenation.[2] String concatenation is represented by a dot.

To avoid ambiguity in variable assignments when constructing a mapping between the left-hand graph of a rule schema and a host graph, we require that expressions in the left graph are *simple*: they (1) contain no arithmetic operators, (2) contain at most one occurrence of a list variable, and (3) do not contain string expressions with more than one occurrence of a string variable. Labels in the right-hand side of a rule schema may contain arithmetic expressions.

The labels of nodes and edges can be *marked* with colours from a fixed set, in addition to a dashed mark for edges only. Marked items match only host graph items with the same mark. There is a special mark `any` that matches arbitrary host graph marks. Nodes with thick borders are *root nodes*. Their purpose is to speed up graph matching, discussed in more detail in the next section.

The programmer can specify a textual condition to add further control to where the rule is applicable, declared by the keyword `where` followed by a boolean expression. GP 2 offers a number of predicates for querying the host graph. For example, the predicate `indeg(1) > 1` in Figure 1 ensures that node 1 is only matched to suitable host graph nodes with more than one incoming edge.

2.2 Fast Rule Schemata

The idea of *rooted* graph transformation [4] is to equip both rule and host graphs with root nodes which support efficient graph matching. Root nodes in rules must match compatible root nodes in the host graph. In this way, the search for a match is localised to the neighbourhood of the host graph's root nodes. It is possible to identify a class of rooted rule schemata that are applicable in constant time if the host graph satisfies certain restrictions.

A conditional rule schema $\langle L \Rightarrow R, c \rangle$ is *fast* if (1) each node in L is reachable from some root (disregarding edge directions), (2) neither L nor R contain repeated 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.

[2] Not to be confused with Haskell's colon operator which adds an element to the beginning of a list.

The first condition ensures that matches can only occur in the neighbourhood of roots. The other conditions rule out linear-time operations, such as copying lists or strings in host graph labels of unbounded length. In [4] it is shown that fast rule schemata can be matched in constant time if there are upper bounds on the maximal node degree and the number of roots in host graphs. The remaining steps of rule application, namely checking the dangling condition and the application condition, removing items from $L - K$, adding items from $R - K$, and relabelling nodes, are achievable in constant time.

2.3 Programs

GP 2 programs consist of a finite number of rule schema declarations and a main command sequence which controls their application order. Execution starts at the top-level procedure `Main`. The user may declare other named procedures, which consist of a mandatory command sequence and optional local rule and procedure declarations. Recursive procedures are not allowed.

The control constructs are: application of a set of conditional rule schemata $\{r_1, \ldots, r_n\}$, where one of the applicable schemata in the set is non-deterministically chosen; sequential composition $P; Q$ of programs P and Q; as-long-as-possible iteration $P!$ of a program P; and conditional branching statements `if` C `then` P `else` Q and `try` C `then` P `else` Q, where C, P and Q are arbitrary command sequences. The meaning of these constructs is formalised with a small-step operational semantics [2].

We just discuss the branching statements. To execute `if` C `then` P `else` Q on a graph G, first C is executed on G. If this produces a graph, then this result is thrown away and P is executed on G. Alternatively, if C fails on G, then Q is executed on G. In this way, graph programs can be used to test a possibly complex condition on a graph without destroying the graph. If one wants to continue with the graph resulting from C, the command `try` C `then` P `else` Q can be used. It first executes C on G and, if this fails, executes Q on G. However, if C produces a graph H, then P is executed on H rather than on G.

3 The GP 2 Compiler

The language is implemented with a compiler, written in C, that translates GP 2 source code to C code. The generated code is executed with the support of a runtime library containing the data structures and operations for graphs and morphisms. We describe how we convert high-level, non-deterministic and rule-based programs into deterministic, imperative programs in C.

3.1 Rule Application

Implementing a graph matching algorithm in the context of graph transformation systems is a well-researched problem. A frequently-used technique is the

```
bool match_rule(morphism *m) {
  return match_n0;
}

bool match_n0(morphism *m) {
  for(root nodes N of the host graph) {
    if(N is not a valid match for n0) continue;
    else {
      flag N as matched;
      update morphism;
      if(match_e0) return true;
    }
  }
  return false;
}

bool match_e0(morphism *m) {
  for(outedges E of match(n0)) {
    if(E is not a valid match for e0) continue;
    else {
      flag E as matched;
      update morphism;
      return true;
    }
  }
  return false;
}
```

Fig. 2: Skeleton of the rule matching code.

search plan, a decomposition of the matching problem into a sequence of primitive matching operations [7]. The compiler supports operations to match an isolated node, to match an edge incident to an already-matched node, and to match a node incident to an already-matched edge. A search plan is constructed by an undirected depth-first traversal of the left-hand side of a rule. When a node or edge is first visited, an operation to match that item is appended to the current search plan. Every iteration of the depth-first search starts at a root node, if one exists, to ensure that the initial "find node" operation is as cheap as possible. If all root nodes have been visited, it starts at an arbitrary unexplored node.

The generated code is a nested chain of matching functions corresponding to the search plan operations. The top-level function is named `match_R` for rule `R`. The pseudocode in Figure 2 illustrates this structure for a rule that matches a root node with a looping edge.

Four checks are made to test if a host graph item h is a valid match for a particular rule item. They are listed below.

1. h is flagged as matched (note that we use injective matching).
2. The rule item is not marked `any` and h's mark is not equal to the rule item's mark.
3. h is not structurally compatible with respect to the rule and the current partial morphism.
4. h's label cannot match the expression of the rule item's label.

The third check differs for nodes and edges. Host graph nodes are ruled out if their degrees are too small. For example, a rule node with two outgoing edges cannot match a host graph node with only one outgoing edge. Host graph edges are checked for source and target consistency. For example, if the target of a rule edge is already matched, the host edge's target must correspond with that node.

To evaluate rule conditions, the compiler writes a function for each predicate and a function to evaluate the whole condition. The predicate functions modify the values of global boolean variables that are queried by the condition evaluator. The condition is checked directly after each call to a predicate function. If the condition is true or all variables in the condition have not been assigned values, matching continues. Otherwise, the match fails and the current matcher returns false, triggering a backtrack. At runtime, the predicate functions are called as soon as they are needed. For example, the function to check the predicate `indeg(1) = indeg(2)` is called immediately after rule node 1 is matched and immediately after rule node 2 is matched. This is done in order to detect an invalid match as soon as possible. To make this possible, a complex data structure is used at compile time to represent conditional rule schemata. The data structure links nodes and variables in the rule to each condition predicate querying that node or variable.

A rule schema contains complete information on the behaviour of the rule, including which items are added, which items are deleted, which items are relabelled, and which variables are required in updated labels. The rule is analysed at compile time to generate code to apply the rule given a morphism. Host graph modifications are performed in the following order to prevent conflicts and dangling edges: delete edges, relabel edges, delete nodes, relabel nodes, add nodes, add edges. The appropriate host nodes, host edges and values of variables are pulled from morphism data structures populated during the matching step.

3.2 Program Analysis for Graph Backtracking

The semantics of GP 2's loop and conditional branching commands require the host graph to be backtracked to a previous state in certain circumstances. For example, the `if-then-else` statement throws away the graph obtained by executing the condition before taking the `then` or `else` branch. Therefore there needs to be a mechanism to preserve older host graph states. We achieve this by maintaining a stack of changes made to the host graph. This is more space-efficient than storing multiple copies of the host graph. This concept is taken from the implementation of the first version of the GP language [11]. At compile time the program text is analysed to determine which portions of the program require

recording of the host graph state. This analysis is quite subtle. For instance, a condition that requires graph backtracking in an `if-then-else` statement may not require graph backtracking in a `try-then-else` statement. We omit the details for lack of space. The first author's PhD thesis [2] describes the program analysis in detail, including the algorithm used by the compiler.

3.3 Program Translation

The main function of the generated C program is responsible for calling the matching and application functions as designated by the command sequence of the GP 2 program. Executing the program amounts to applying a sequence of rules. The code generator writes a short code fragment for each rule call and translates each control construct into an equivalent C control construct. The runtime code is supported by a number of global variables, including the host graphs and morphisms. A global boolean variable *success*, initialised to true, stores the outcome of a computation to support the control flow of the program.

A standard rule call generates code trying to match the rule. If a match is found, the code calls the rule application function and sets the success flag to true. If not, control passes to failure code. Certain classes of rules allow simpler code to be generated. For example, a rule with an empty left-hand side does not generate code to call a matching function. The failure code is context-sensitive. If there is a failure at the top level, the program is terminated after reporting to the user and freeing memory. Failure in a condition guard sets the global success flag to false so that control goes to the else branch of the conditional statement. Failure in a loop sets the success flag to false and calls the function `undoChanges` (described below) to restore the host graph to the state at the start of the most recent loop iteration.

Figure 3 summarises the translation of some GP 2 control constructs to C. The rule set call {`R1, R2`} is tackled by applying the rules in textual order until either one rule matches or they all fail. The do-while loop is used to exit the rule set if a rule matches before the last rule has been reached. The condition of a branching statement is executed in a do-while loop: if failure occurs before the last command of the condition, the break statement is used to exit the condition, and control is assumed by the then/else branch. GP 2's loop translates directly to a C while loop. One subtlety is the looped command sequence, where the line `if(!success) break;` is printed after the code for all commands except the last. A second subtlety is that success is set to true after exiting a loop because GP 2's semantic rules state that a loop cannot fail. Command sequences are handled by generating the code for each command in the designated order. When a procedure call is encountered in the program text, the code generator inlines the command sequence of the procedure at the point of the call.

Restore points (the variables named *rp* in Figure 3) are created and assigned to the top of the graph change stack when graph backtracking is required. The function *undoChanges* restores a previous host graph state by popping and undoing changes from the stack until the restore point is reached. The function *discardChanges* pops the changes but does not undo them. It is only called at

Command	Generated Code

`{R1, R2}`	`do {`
	`  if(matchR1(M_R1)) {`
	`  <success code>`
	`  break;`
	`  }`
	`  if(matchR2(M_R2)) <success code>`
	`  else <failure code>`
	`} while(false)`

`if C then P`
`else Q`
	`int rp = <top of GCS>;`
	`do C while(false);`
	`undoChanges(host, rp);`
	`if(success) P else Q;`

`try C then P`
`else Q`
	`int rp = <top of GCS>;`
	`do C while(false);`
	`if(success) P`
	`else {`
	`  undoChanges(host, rp);`
	`  Q`
	`}`

`(P; Q)!`	`int rp = <top of GCS>;`
	`while(success) {`
	`  P`
	`  if(!success) break;`
	`  Q`
	`  if(success) discardChanges(rp);`
	`}`
	`success = true;`

Fig. 3: C code for GP 2 control constructs

the end of a successful loop iteration to prevent a failure in a future loop iteration from causing the host graph to roll back beyond the start of its preceding iteration. Each restore point has a unique identifier to facilitate multiple graph backtracking points.

The compiler respects the formal semantics of GP 2 (given in [13] and in updated form in [2]) in that any output graph of the generated code is admissible by the semantics. Similarly, a program run ending in failure is possible only if the semantics allows it. We did not formally prove this kind of soundness—that would be a tremendous project far beyond the scope of this work. Also, there is no guarantee that a program run terminates if a terminating execution path exists (this would require a breadth-first strategy which is impractical).

3.4 Runtime Library

The runtime library is a collection of data structures and operations used by the generated code during rule matching, rule application and host graph backtracking. As aforementioned, graph backtracking is performed by a graph change stack. We describe the other core data structures of the runtime library.

The host graph structure stores node and edge structures in dynamic arrays. Free lists are used to prevent fragmentation. Nodes and edges are uniquely identified by their indices in these arrays. The graph structure also stores the node count, the edge count, and a linked list of root node identifiers for fast access to the root nodes in the host graph. A node structure contains the node's identifier, its label, its degrees, references to its inedges and outedges, a root flag, and a matched flag. An edge structure contains the edge's identifier, its label, the identifiers of its source and target, and a matched flag.

A label is represented as a structure containing the mark (an enumerated type), a pointer to the list and the length of the list. GP 2 lists are represented internally as doubly-linked lists. Each element of the list stores a type marker and a union of integers and strings, equivalent to GP 2's atom type. Lists are stored centrally in a hash table to prevent unnecessary and space-consuming duplication of lists for large host graphs with repeated labels.

The morphism data structure needs to capture the node-to-node and edge-to-edge mappings, and the assignment mapping variables to their values. Thus the data structure used to represent morphisms contains the following four substructures: (1) an array of host node identifiers, (2) an array of host edge identifiers, (3) an array of assignments, and (4) a stack of variable identifiers. At compile time each node, edge and variable in a rule is identified with an index of its array in the morphism, allowing quick access to the appropriate elements. The stack is used to record assignment indices in the order in which the variables are assigned values. This is needed because the variables encountered at runtime are not guaranteed to agree with the compile-time order.

4 Case Study: 2-Colouring

Vertex colouring has many applications [18] and is among the most frequently considered graph problems. We focus on 2-colourability: a graph is *2-colourable*, or *bipartite*, if one of two colours can be assigned to each node such that the source and target of each non-loop edge have different colours.

Figure 4 shows a rooted GP 2 program for 2-colouring. The input is a connected, unmarked and unrooted graph G. If G is bipartite, the output is a valid 2-colouring of G. Otherwise, the output is G. The edges in this program are *bidirectional* edges, graphically denoted by lines without an arrowhead. Such a rule matches a host graph edge incident to two suitable nodes independent of the edge's direction. (This is syntactic sugar: a rule with one bidirectional edge is equivalent to a rule set containing two rules with the edge pointing in different directions.) The rules `colour_red` and `joined_blues` are omitted, which are the

"inverted" versions of the rules `colour_blue` and `joined_reds` with respect to the node marks. In particular, the right-hand side of `joined_blues` also has a grey root node.

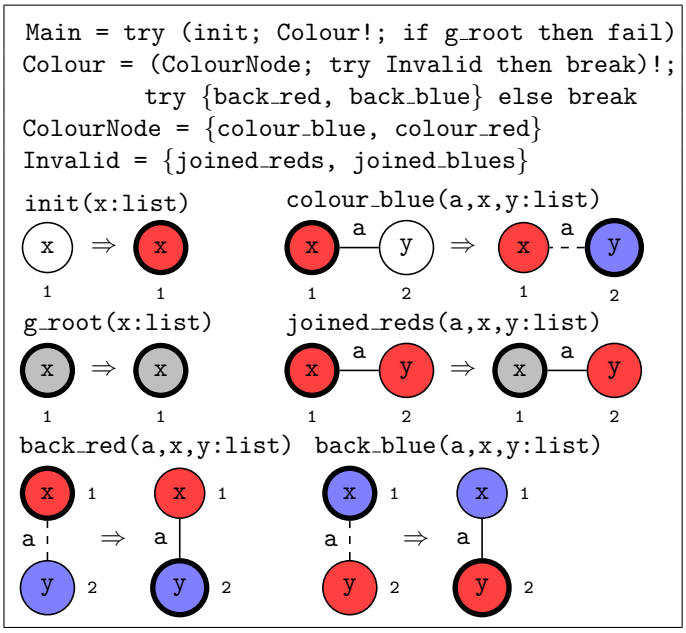


Fig. 4: The program `2colouring`

At its core, `2colouring` is an undirected depth-first traversal in which the source node is chosen non-deterministically. The root node represents the current position in the traversal. The rule `init` prepares the search by matching an arbitrary host graph node, making it the root node, and colouring it red. Each iteration of the `Colour!` loop does the following:

1. `ColourNode`: move the root node to an adjacent uncoloured node and colour it with the opposite colour. Dash the edge connecting the current root node to the previous one.
2. `try Invalid else break`: check if the current root node is adjacent to any nodes with the same colour. If so, mark the root node grey and break the inner loop.
3. Repeat steps (1) and (2) until no more rules are applicable.
4. `try {back_red, back_blue} else break`: move the root along a dashed edge and undash the edge. If this is not possible, break the outer loop.

Observe that the dashed edges act as a "trail of breadcrumbs" to facilitate backtracking. If the 2-colourability is violated at any point during the computation, the root node is marked grey, which acts as a flag for non-bipartiteness.

Once the `Colour!` exits, the remainder of the program (`if g_root then fail`) checks if the root node is grey. If the root node is grey, then the `fail` command causes the `try-then-else` to take the `else` branch, and the host graph assumes its state before entering the branch, which returns the input graph. Otherwise, the `then` branch is taken, which returns the current 2-coloured graph.

Termination is guaranteed because each rule either decreases the number of unmarked nodes or decreases the number of dashed edges while preserving the number of unmarked nodes. Therefore, at some point, a back rule will fail because there exist no dashed edges, or a colouring rule will fail because there exist no unmarked nodes.

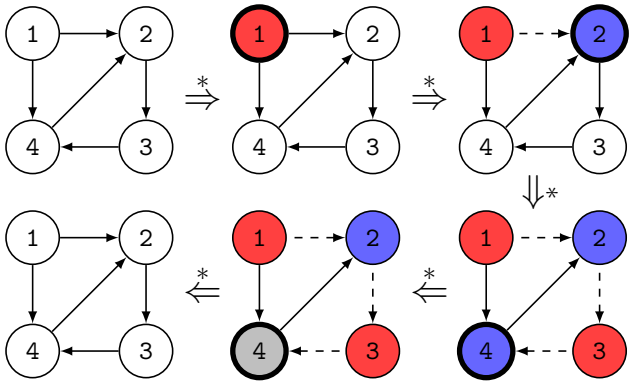


Fig. 5: Example run of `2colouring`

Figure 5 shows the execution of `2colouring` on the host graph in the upper-left of the diagram. This graph is clearly not 2-colourable. The rule `init` colours node 1 red. The rule `colour_blue` nondeterministically matches the edge $1 \to 2$. It roots node 2, colours it blue and dashes the edge. The colouring rules are applied twice more to give the lower-right graph. At this point the rule `joined_blues` matches the edge $4 \to 2$. This colours the root node grey. The inner loop breaks, and control passes to `Backtrack`. Both back rules fail because neither match a grey root node. This causes the outer loop to break. Finally, `g_root` succeeds, causing the `try` statement to fail and return the original graph.

The following result, proved in [2], assumes that input graphs are unmarked and connected.

Proposition 1 (Time complexity of `2colouring`). *On graphs with bounded node degree, the running time of `2colouring` is linear in the size of graphs. On unrestricted graphs, the running time is quadratic in the size of graphs.*

Here "size of graphs" refers to the number of nodes and edges in host graphs. The result is independent of the size of host graph labels.

5 Performance

To experimentally validate the time complexity of `2colouring`, and to test the performance of the language implementation, we ran the generated C code for `2colouring` against an adaptation of Sedgewick's hand-crafted C program for 2-colouring [17].

We chose two classes of input graphs. The first class is *square grids* (abbreviated *grids*), which are suitable because: (1) grids are 2-colourable. This guarantees that both programs perform the same computation, namely matching and colouring every node in the graph; (2) grids have bounded node degree, which tests the linear complexity of `2colouring`; (3) it is relatively simple to generate large grids. The second class is *star graphs*, used to test the performance on graphs of unbounded degree. A star graph consists of a central node with k outgoing edges. The targets of these outgoing edges themselves have a single outgoing edge. Star graphs share properties (1) and (3) of grid graphs. Examples can be seen in Figure 6.

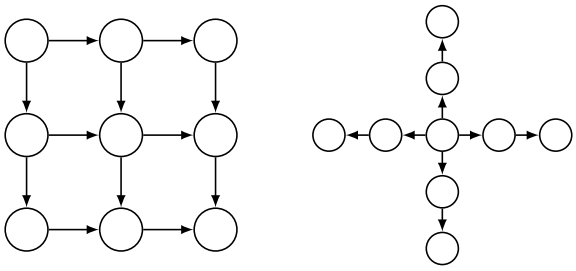


Fig. 6: Examples of a square grid graph and a star graph

5.1 2-colouring in C

This section describes a C implementation of 2-colouring based on the code in Sedgewick's textbook *Algorithms in C* [17] which uses an adjacency list data structure for host graphs. For a graph with n nodes, an adjacency list is a node-indexed array containing n linked lists. An edge $i \to j$ is represented by the presence of j in the ith linked list, and vice versa if the graph is undirected. For our purposes there is no requirement to implement a graph data structure that supports the complete GP 2 feature set. Instead, we exploit some of the properties of the algorithms and host graphs we wish to execute in order to develop a minimal graph data structure.

We adapt Sedgewick's adjacency-list data structure and functions for host graphs. The main graph structure stores counts of the number of nodes and edges, a node-indexed array of adjacency lists, and a node-indexed array of integer node labels. Adjacency lists are represented internally by linked lists,

```
1  bool dfsColour(int node, int colour) {
2     Link *l = NULL;
3     int new_colour = colour == 1 ? 2 : 1;
4     host->label[node] = new_colour;
5     for(l = host->adj[node]; l != NULL;
6         l = l->next)
7        if(host->label[l->id] == 0) {
8           if(!dfsColour(l->id, new_colour))
9              return false;
10       }
11       else if(host->label[l->id] != colour)
12          return false;
13    return true;
14 }
15
16 int main(int argc, char **argv) {
17    host = buildHostGraph(argv[1]);
18    bool colourable = true;
19    int v;
20    for(v = 0; v < host->nodes; v++)
21       if(host->label[v] == 0)
22          if(!dfsColour(v, 1)) {
23             colourable = false; break;
24          }
25    if(!colourable)
26       // Unmark all nodes.
27       for(v = 0; v < host->nodes; v++)
28          host->label[v] = 0;
29    return 0;
30 }
```

Fig. 7: DFS 2-colouring in C

where each list element stores the node identifier of a target of one of its outgoing edges.

At runtime, the GP 2 compiler's host graph parser is used to read the host graph text file and construct the graph data structure. This minimises the gap between the handwritten C code and the code generated from the GP 2 compiler, so that the comparison between the performance of the actual computations on the host graph is as fair as possible.

The C algorithm for 2-colouring is given in Figure 7. Code for error checking, host graph building, and declaration of global variables is omitted. The program takes a single command line argument: the file path of the host graph. The function `buildHostGraph` initialises and adds edges to the graph (via the global Graph pointer *host*) through the GP 2 host graph parser.

Nodes are labelled 0, 1 or 2. Node labels are initialised to 0, representing an uncoloured and unvisited node. 1 and 2 represent the two colours of the algorithm. The function `dfsColour` is called recursively on all uncoloured nodes of the host graph. It is passed a node v and a colour c as its argument. It colours v with the contrasting colour c', and goes through v's adjacency list. If an adjacent node is uncoloured, `dfsColour` is called on that node. If an adjacent node is also coloured c', the function returns false, which will propagate through its parent calls and to the main function. If main detects a failure (line 25), it sets the label of all nodes to 0 and exits. Otherwise, the coloured graph is returned.

Figure 8 show the comparison of runtimes of both programs. There is almost no difference between the time it takes for either program to 2-colour grids, a remarkable result considering the compiled GP 2 code explicitly performs (rooted) subgraph matching at each step, while the tailored C program navigates a simple pointer structure. However, the star graph plot makes it clear that tailored C code is not limited by bounds on node degree. The compiled GP 2 code displays quadratic time complexity because it searches the outgoing edge list of the central node in the same order for every rule match.

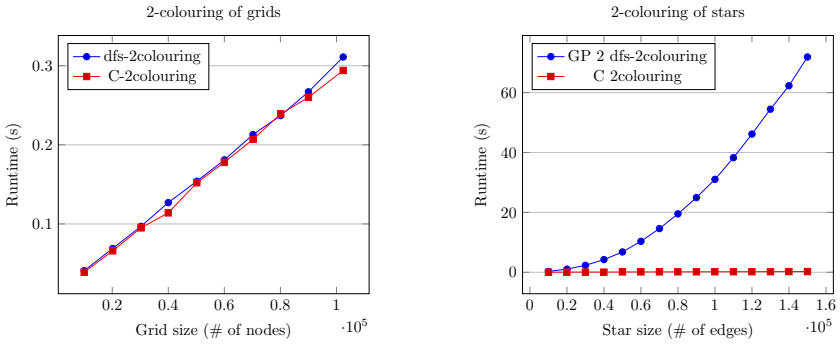


Fig. 8: Plots of the runtimes of the 2-colouring programs in GP 2 and C

6 Related Work

There exist a number of tools and languages for programming with graph transformation rules, including AGG [15], GROOVE [9] and PORGY [8]. We highlight three implementations with code generation. PROGRES [16] generates efficient Modula-2 or C code from transformation specifications. The code generator is more complex than that of GP 2 because it must handle sophisticated language features, for example arbitrary path expressions in rules and derived attributes. Programs in GrGEN.NET [10] are compiled to highly-optimised .NET assemblies for high performance execution. The code generator of the model transformation tool GReAT [19] has some similarity to that of GP 2: both generate pattern

matching code that searches the host graph with user-declared root points to prune the search space. However, there are some differences because of the different feature sets of the languages. For example, GReAT's code generator must handle passing matched objects from one rule to another, while GP 2's code generator must handle application conditions during graph matching.

The concept of rooted rules has been used in various forms in implementations of graph transformation. To mention a couple of examples, rules in GrGEN.NET and GReAT can return graph elements to restrict the location of subsequent rule applications [10, 1], and the strategy language of PORGY restricts matches of rules to a subgraph of the host graph called the position which can be transformed by the program [8].

7 Conclusion and Future Work

We have reviewed the visual programming language GP 2 based on graph transformation rules and described a compiler that translates high-level GP 2 programs to C code. A novel aspect of our implementation is generating search plans at compile time and using them to systematically generate structured and readable C code. Another distinctive feature is the static analysis of programs to determine if code needs to be generated to facilitate the recording of the host graph state. Using the compiler, we show that the generated C code for a depth-first 2-colouring program performs as quickly as a handcrafted C program also based on depth-first search on a class of host graphs with bounded node degree. These initial results are good, but more case studies ought to be investigated to further demonstrate the efficiency of the generated code, in particular programs that transform the host graph structurally, such as a reduction program to identify membership in a specific graph class.

A limitation of the GP 2 implementation is that it makes little effort to optimise rule matching for rules without root nodes. One method of speeding up matching is to compute optimal search plans at runtime based on an analysis of the host graph. This has been implemented in GrGEN.NET [5]. Another approach is to optimise rule matching at compile time. An example of such an optimisation is transforming a looped rule call to code that finds all matches in the host graph and performs the modifications in one step, which in general is more efficient than finding one match and starting a new search for the next match. This requires some care because pairs of matches could be in conflict.

References

1. A. Agrawal, G. Karsai, S. Neema, F. Shi, and A. Vizhanyo. The design of a language for model transformations. *Software and System Modeling*, 5(3):261–288, 2006.
2. C. Bak. *GP 2: Efficient Implementation of a Graph Programming Language*. PhD thesis, Department of Computer Science, The University of York, 2015.

3. C. Bak, G. Faulkner, D. Plump, and C. Runciman. A reference interpreter for the graph programming language GP 2. In *Proc. Graphs as Models (GaM 2015)*, volume 181 of *Electronic Proceedings in Theoretical Computer Science*, pages 48–64, 2015.
4. C. Bak and D. Plump. Rooted graph programs. In *Proc. International Workshop on Graph-Based Tools (GraBaTs 2012)*, volume 54 of *Electronic Communications of the EASST*, 2012.
5. G. V. Batz, M. Kroll, and R. Geiß. A first experimental evaluation of search plan driven graph pattern matching. In *Applications of Graph Transformations with Industrial Relevance, Revised Selected and Invited Papers*, volume 5088 of *Lecture Notes in Computer Science*, pages 471–486. Springer, 2007.
6. M. Dodds and D. Plump. Graph transformation in constant time. In *Proc. International Conference on Graph Transformation (ICGT 2006)*, volume 4178 of *Lecture Notes in Computer Science*, pages 367–382. Springer, 2006.
7. H. Dörr. *Efficient Graph Rewriting and its Implementation*, volume 922 of *Lecture Notes in Computer Science*. Springer, 1995.
8. M. Fernández, H. Kirchner, I. Mackie, and B. Pinaud. Visual modelling of complex systems: Towards an abstract machine for PORGY. In *Proc. Computability in Europe (CiE 2014)*, volume 8493 of *Lecture Notes in Computer Science*, pages 183–193. Springer, 2014.
9. A. H. Ghamarian, M. de Mol, A. Rensink, E. Zambon, and M. Zimakova. Modelling and analysis using GROOVE. *International Journal on Software Tools for Technology Transfer*, 14(1):15–40, 2012.
10. E. Jakumeit, S. Buchwald, and M. Kroll. GrGen.NET - the expressive, convenient and fast graph rewrite system. *Software Tools for Technology Transfer*, 12(3–4):263–271, 2010.
11. G. Manning and D. Plump. The GP programming system. In *Proc. Graph Transformation and Visual Modelling Techniques (GT-VMT 2008)*, volume 10 of *Electronic Communications of the EASST*, 2008.
12. D. Plump. The graph programming language GP. In *Proc. Algebraic Informatics (CAI 2009)*, volume 5725 of *Lecture Notes in Computer Science*, pages 99–122. Springer, 2009.
13. D. Plump. The design of GP 2. In *Proc. International Workshop on Reduction Strategies in Rewriting and Programming (WRS 2011)*, volume 82 of *Electronic Proceedings in Theoretical Computer Science*, pages 1–16, 2012.
14. C. M. Poskitt and D. Plump. Hoare-style verification of graph programs. *Fundamenta Informaticae*, 118(1-2):135–175, 2012.
15. O. Runge, C. Ermel, and G. Taentzer. AGG 2.0 – new features for specifying and analyzing algebraic graph transformations. In *Proc. Applications of Graph Transformations with Industrial Relevance (AGTIVE 2011)*, volume 7233 of *Lecture Notes in Computer Science*, pages 81–88. Springer, 2011.
16. A. Schürr, A. Winter, and A. Zündorf. The PROGRES approach: Language and environment. In H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors, *Handbook of Graph Grammars and Computing by Graph Transformation*, volume 2, chapter 13, pages 487–550. World Scientific, 1999.
17. R. Sedgewick. *Algorithms in C: Part 5: Graph Algorithms*. Addison-Wesley, 2002.
18. S. S. Skiena. *The Algorithm Design Manual*. Springer, second edition, 2008.
19. A. Vizhanyo, A. Agrawal, and F. Shi. Towards generation of efficient transformations. In *Proc. Generative Programming and Component Engineering (GPCE 2004)*, volume 3286 of *Lecture Notes in Computer Science*, pages 298–316. Springer, 2004.