The University of York Department of Computer Science

Submitted in part fulfilment for the degree of MEng.

Tracing and Debugging in GP2

Joshua Asch

Date TBC

Supervisor: Detlef Plump

Number of words = 0, as counted by `wc -w`.
This includes the body of the report only.

Abstract

This is my project!

Contents

List of Figures

List of Tables

1 Introduction

1.1 Introductiony Bit

A section introducing the project.

1.2 Ethics

A section discussing the ethics of the project.

2 Literature Review

2.1 Programming by Graph Transformation

2.1.1 Graph Programming

Graph programming involves a series of transformations applied to a graph. The problem being solved must be redefined in terms of a start graph and an algorithm represented by graph transformations. The final graph at the end of the algorithm gives the solution to the problem.

Historically, programming by graph transformation required using a programming language such as C or Java, implementing data structures to represent graphs, and directly making modifications to the graph in the program. However, recently some attempts have been made to create tools for graph programming which abstract away the representation of the graphs, allowing the programmer to focus on the program itself.

Some of these tools include PROGRES [1], AGG [2], GrGen [3], and, most recently, GP2 [4]. All of these are domain-specific languages for graph programming which also provide a graphical interface to describe graphs and transformations.

These kinds of tools take a representation of a graph program, as defined in their graphical editor, and transform this into a runnable program. This can be implemented in Java (in the case of AGG and GROOVE) or in C (in the case of PROGRES and GP2). This program can then be executed to find the output graph generated by the algorithm.

2.1.2 The GP2 Language

GP2 (Graph Programs 2) is a programming language developed at the University of York [4, 5], an updated implementation of the original language, GP [6]. It is designed for writing programs at a high level, to perform graph transformations without having to implement data structures to represent the graphs in more traditional lower level languages such as C.

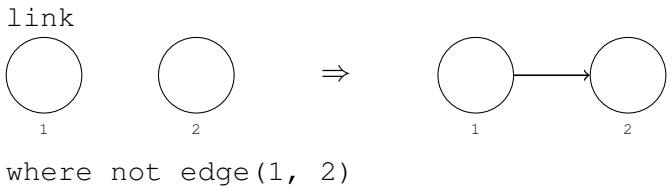

Figure 2.1: A rule in GP2

Programming in GP2 consists of an input graph, known as the *host graph*, a set of *rules*, and a *program* which defines the order in which to apply the rules. Running a GP2 program on a host graph produces a new graph as a result, called the *output graph*.

Rules

Rules are the basic building blocks of a GP2 program and are defined by a left-hand-side (LHS), a right-hand-side (RHS), and optionally a conditional clause. A rule can be thought of as the definition of a transformation; a subgraph matching the LHS of the rule is transformed to resemble the RHS. An example of a GP2 rule is shown in Figure 2.1.

The conditional clause is used to specify additional constraints on the subgraph matching the LHS. Any match has to both match the LHS and conform to the constraints defined by the conditional clause.

In a compiled program, a rule is split into two phases. The *match* phase searches the current graph for a subgraph which matches the LHS of the rule. This must be an *injective match*, meaning that each item in the LHS must be matched with exactly one item in the host graph.

In this implementation of GP2, rule matches are chosen determinstically due to the impracticality of generating all non-deterministic possibilities, which is what GP1 did. If no match is found for the LHS, the rule is considered *failed*. If a match is found, the program moves on to the second phase, the *application*.

A rule specifies a number of *interface nodes*, nodes which are present in both the LHS and the RHS. These interface nodes can be seen in Figure 2.1; they are the nodes with small numeric labels. During the application phase, any nodes in the LHS which are *not* interface nodes will be deleted. Similarly, any non-interface nodes which appear in the RHS will be created. At the end of the application phase, the subgraph

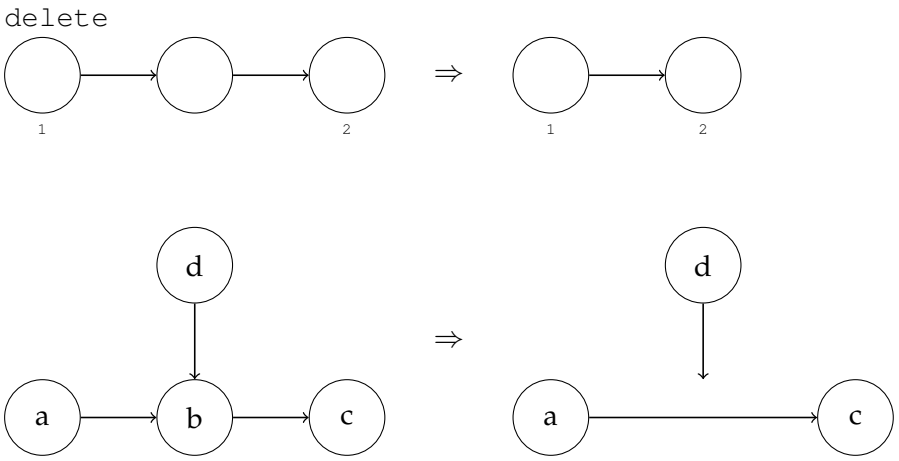

Figure 2.2: Attempting to apply a rule when the match violates the dangling condition

will match the RHS of the rule definition. The new graph created by the application of this rule, an *intermediate graph*, is then used as the input to the next part of the program.

In the example in Figure 2.1, the program will search for a subgraph containing two nodes without an edge connecting them. If a match is found, it will be transformed to resemble the RHS by adding an edge between the nodes.

The Dangling Condition

There is an additional condition which determines whether a rule can be matched, the *dangling condition*. This condition states that if a rule causes a node to be deleted, the rule must also delete all edges incident to that node. In other words, all non-interface nodes in a rule match must have no incoming or outgoing edges which are not also part of the match.

Figure 2.2 shows the result of attempting to apply a rule where the match violates the dangling condition. Since node b is not an interface node, it is deleted during the application of the `delete` rule. However, the rule does not specify that the edge from b to d should be deleted. This leaves a *dangling edge* from node d with no target. The dangling condition is used to ensure that rules result in no dangling edges.

Programs

A GP2 program defines the order in which to apply rules using 8 simple control structures:

SEQUENCE Two subprograms separated by a semicolon "`P; Q`" are applied one after the other.

RULE SET Rules in curly braces "`{`R_1`, `R_2`, `R_3`}`" define a set, where exactly one rule from the set is executed, unless no rule in the set can be matched. The implementation of GP2 selects a rule deterministically, though the the specification allows non-determinism.

IF-THEN-ELSE In the statement "`if C then P else Q`", the sub-program `C` is executed, and the result, i.e. success or failure, is recorded, before reverting any changes caused by C. Then, if C succeeded, `P` is executed on the original graph. If it failed, then `Q` is executed on the original graph. Note that by taking a copy first, any changes made by `C` are reverted before executing either `P` or `Q`.

TRY-THEN-ELSE Similar to IF-THEN-ELSE, but `C` is only reverted if it fails. Thus any changes made by `C` are *not* reverted before executing `P`, but they *are* reverted before executing `Q`.

AS-LONG-AS-POSSIBLE A subprogram followed by an exclamation point "`P!`" is matched and applied repeatedly until it fails. When `P` fails, it does not transition the program to the fail state.

PROCEDURE Similar to a C preprocessor macro, a procedure is simply a named subprogram where any reference to the procedure name can be replaced with the definition of the procedure.

SKIP A no-op which always succeeds, and does not affect the graph. Invoked using the keyword "`skip`".

FAIL A no-op which always fails and does not affect the graph. This is the same as attempting to execute a rule for which there are no matches. Invoked using the keyword "`fail`".

For GP2, a subprogram is either a single rule, referenced by its name, or one of the above control structures. Therefore it is possible to nest control structures to create more complex programs.

In general, execution of a program continues until either all statements are executed, or until a statement results in an attempt to apply a rule which has no matches in the graph. The exceptions to this are As-Long-As-Possible statements, and the conditional statements in If-Then-Else and Try-Then-Else structures. In these cases, a failure to match a rule does not halt execution of the program.

Figure 2.3 shows an example GP2 program, an adapted version of the program used as a case study in Bak's thesis on GP2 [5, p.126]. It is a simple program which determines whether a graph is *2-colourable*, that is, its nodes can be coloured using two different colours without two nodes of the same colour being connected by an edge.

This program consists of four rules and uses many of the constructs outlined previously, including Try-Then-Else, If-Then-Else, Rule Sets, As-Long-As-Possible and Procedures.

It also takes advantage of another feature of GP2, the ability to *colour* a node. The colour of a node is a property which can be set by a rule, or the host graph can contain coloured nodes. The colour is taken into account when searching for a rule match; for example, in this program, `joined_reds` will only match when both the nodes' colours are red.

An example execution of the 2-colouring program is shown in Figure 2.4. Starting with an uncoloured graph, the algorithm picks a node and colours it red using the `init` rule. It then traverses the graph colouring nodes in alternating colours using the `colour_blue` and `colour_red` rules, by defining them as a Rule Set in a Procedure and executing it As-Long-As-Possible. When no more uncoloured nodes are present in the graph, the `Colour` procedure will be unable to match any further rules, so it will end.

To check whether the produced colouring is valid, the entire `Main` procedure is wrapped in a Try-Then-Else statement. After executing `Colour`, the `Invalid` procedure runs. This procedure uses the two remaining rules, `joined_reds` and `joined_blues`, to see if any adjacent nodes are the same colour. If they are, one of these rules will match, triggering the conditional statement `fail` from the If-Then-Else statement. This in turn causes the outer `try` to fail, reverting all changes made to the graph and returning the uncoloured input graph.

However, if `Invalid` fails to match either of the rules, it must mean that no two same-coloured nodes are connected via an edge. This means that it is a valid colouring. The `fail` statement is not executed, meaning the `try` succeeds. The changes to the graph are kept, and the modified graph is returned as the result of the program.

```
Main = try (init; Colour!; if Invalid then fail)
Colour = {colour_blue, colour_red}
Invalid = {joined_reds, joined_blues}
```

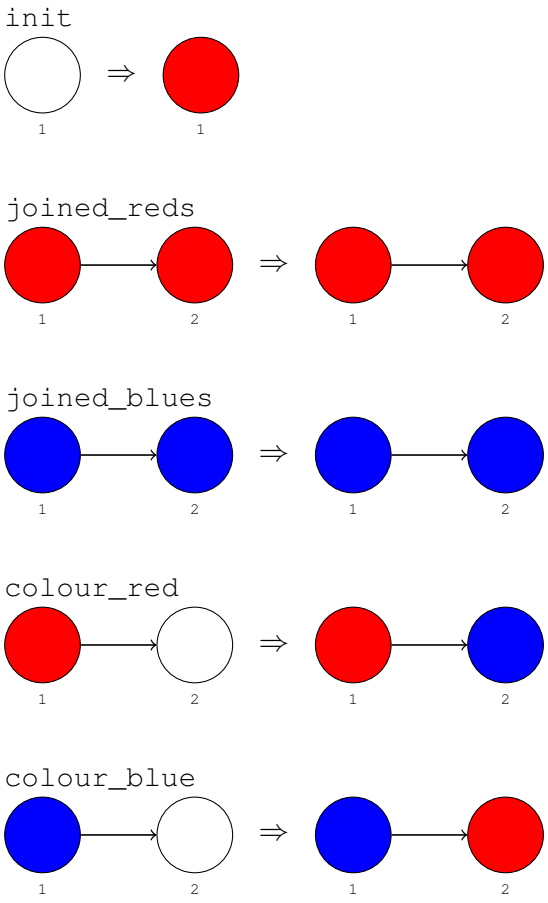

Figure 2.3: Definition of 2-colouring in GP2

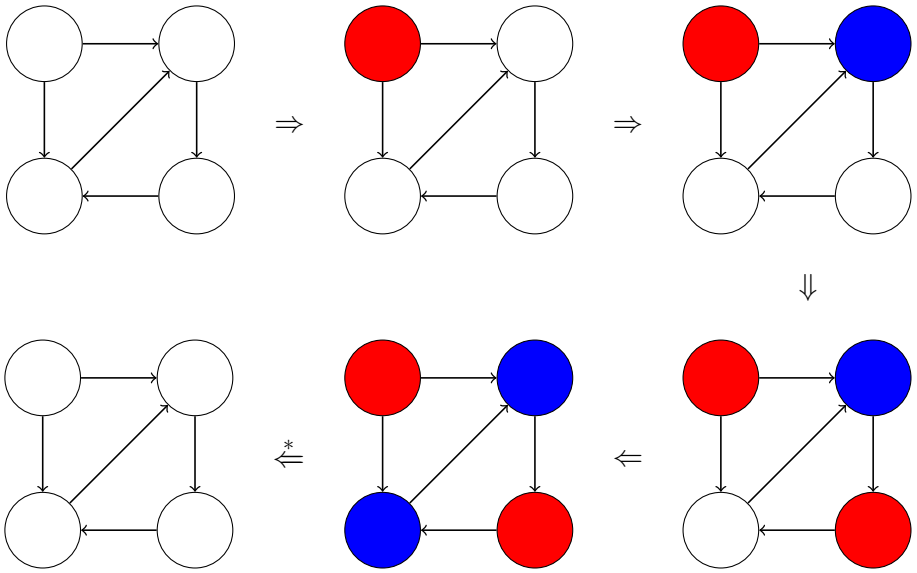

Figure 2.4: Example execution of 2-colouring

In the example execution in Figure 2.4, the host graph is not 2-colourable. When the rules in `Colour` can no longer be matched, `joined_blues` inside `Invalid` will match with the nodes in the bottom-left and top-right of the graph. Because this rule was successfully applied (even though it has no effect on the graph), the If-Then-Else causes the program to enter the fail state. This in turn means that the Try-Then-Else surrounding the program fails, and all the changes made to the input graph are reversed. The output graph is therefore identical to the input graph.

2.2 Tracing and Debugging

2.2.1 Debugging in Imperative Languages

When programming in a high level imperative language, such as C or Java, the programmer will more often than not have access to a *debugger*. In [7], Sammet says that a program in a high level language without debugger support may be "harder to debug than an assembly language which the programmer understands", since the programmer has to interpret the compiled code, rather than reading the source code they wrote.

Most language interpreters and Integrated Development Environments (IDEs) have a debugger built in [8], and standalone debuggers are also available for some languages. For C, this may be `gdb` [9], while for Java, it might be `jdb` [10].

A debugger is intended to allow the programmer to pause their program during execution, so that they can inspect the contents of variables and other memory locations. It also allows them to run their program step-by-step to see its execution flow; they may wish to check that a function is called at the expected point during execution, for instance.

Some debuggers also include more advanced features to make debugging easier and to give the programmer more insight into their program. Breakpoints are a common feature which allow the programmer to specify a line of source code and have the program execute normally until the breakpoint is reached, at which point execution will pause, or *break*.

Because GP2 has a rule-based structure, unlike imperative programming languages, it is not clear what a "line of code" represents in a GP2 program. The textual part of a GP2 program can indeed have lines, but since a single line in a GP2 program can be complex and use many rules, this is too high-level to consider a "line of code" as a debugger does. In an imperative language, a single line of code may only perform one or two actions, so inspecting each line individually shows the flow of the program. In GP2, running a line of the textual program could result in tens or hundreds of rules being applied if As-Long-As-Possible is used, for example. It may be inappropriate to focus on how debuggers are implemented for imperative languages when considering GP2.

IDEs

A debugger is often available from within the Integrated Development Environment (IDE) for a language. For example, the Visual Studio IDE for C, C++, and C# includes the Visual Studio Debugger [11]. One of the most prevalent Java IDEs, Eclipse, integrates with `jdb` [12].

When an IDE integrates with a debugger, it can provide additional functionality by allowing the programmer to interact with the source code and the debugger visually in the same environment. Visual Studio and Eclipse both allow breakpoints to be set directly on a line of source code in the editor, for instance. IDEs often allow the programmer to trace program execution through the source code, by highlighting each line of code as the programmer steps through in the debugger.

It may be useful to use some of these ideas for GP2. For instance, when

stepping through a GP2 program, it would be helpful to highlight the current statement or rule being executed, to give the user context.

Edit-and-Continue

Edit-and-continue is an even more advanced feature which requires specific compiler support, and is usually only available in IDEs, since they have access to both the compiler and the debugger. It allows the programmer to pause execution of the program, edit the source code, recompile the program, and continue execution from the previous paused state, without having to restart the program from the beginning. Edit-and-continue is useful for reducing the time taken to find and fix bugs, since fixes can be implemented and tested without having to stop and restart the program's execution.

It is unclear whether edit-and-continue would be a good addition to GP2. On one hand, it can speed up debugging because the programmer does not have to start the program again from the beginning. On the other hand, because GP2 is such a high level language, it may not make sense to resume a program that has been edited. There is a high chance that the modifications would affect the algorithm enough that the current state would not be reached if the program was restarted from the beginning.

Reverse Debugging

`gdb` supports what is called *reverse debugging* [13]. This allows program execution to actually be reversed, running the program backwards to reach an earlier state. This can come in useful to look for non-deterministic bugs which do not always occur; the program can be run until the bug occurs, then executed in reverse to look for the cause.

This ability comes with a trade-off, however; running with reverse debugging enabled reduces the performance of the running program. It can only be used in specific cases and cannot be enabled all the time, since the program would run much slower and possibly exhibit time-related bugs. Reverse debugging is also only available for `gdb` running on Linux.

`gdb`'s implementation of reverse debugging involves recording the machine state after each instruction exectuion, including the values stored in memory and registers. To reverse an instruction, the state from the previous instruction is simply restored, making it appear as if the reversed instruction was never executed. This implementation allows powerful interaction with the program; it can reverse a single instruction at a

time, or it can be run backwards until a breakpoint is reached. In theory, although `gdb` does not support this, this system could allow a form of "checkpointing" where execution can be skipped directly back to an arbitrary point by simply restoring the state from that point.

Reverse debugging may be a useful feature for GP2, because it would allow a programmer to step backwards through the execution of a GP2 program to inspect the intermediate graph before and after a rule application, to see what changed. This method is more akin to tracing than traditional debugging, since the debugger is storing a history of everything that occurs during execution.

While the implementation of reverse debugging in `gdb` is limited to the Linux platform, this is because its implementation requires monitoring calls to the kernel and system APIs. This is irrelevant to GP2, because it would only need to trace rule matches and applications, both of which are implemented inside GP2 itself.

2.2.2 Tracing in Functional Languages

The syntax and execution of a GP2 program is very similar to that of a functional programming language. For example, a function in Haskell is similar to a rule or set of rules in GP2; the function defines left-hand-sides and right-hand-sides, and executing it with an argument looks for a LHS which matches the argument, and returns the RHS. This similarity to the matching and application of rules in GP2 suggests that it would be appropriate to consider it like a functional language when implementing debugging or tracing.

Because of the nature of functional languages, it is rare to see traditional debuggers like those used with imperative languages [14]. Lazy evaluation, where the value of a statement is only calculated when it is required, means that pausing execution on a line of code may not reveal the value of a statement on that line, because it will not be evaluated until later in the program.

To avoid this problem, functional programmers will often use tracing instead. This is where additional code is added to the program which simply outputs information about what the program is doing, either to the console or to a file on disk. The progammer then reads this information back once the program has finished, to see what steps the program took and identify where it differed from the expected execution.

Tracing can be done in primitive ways, by manually adding `print` statements to the code, but there are also more sophisticated tools avail-

able. For Haskell, for instance, a handful of different tracing tools are available [15]. Two of them, Freja and Hat, are modified Haskell compilers which add automatic tracing functionality to the compiled program. When the program runs, trace information is stored in the program heap, which can then be accessed at the end of execution. The other tool, Hood, is a Haskell library which is used by importing it and adding `observe` annotations to the code, which preserve lazy evaluation and output information about the program as it runs. The programmer would then inspect the output from the annotations to trace through the program.

The benefit of compiler based tools like Freja and Hat is the programmer does not need to think about where to put tracing code, since all code is traced automatically in the compiled program. However, storing a full trace of an entire program takes up space either in memory or on disk; the advantage of a manual tool like Hood is the programmer can reduce the number of trace points to reduce the trace size.

2.2.3 Previous Work on Debugging in GP2

There has been some previous work to add debugging facilities to GP2 [16]. This work was focused on modifying the compiler to support stepping through a program.

With the new compiler, command line arguments were added to allow the user to specify how big a "step" should be, and how many steps to execute before stopping. This allows GP2 to approximate breakpoints by picking an appropriate step size and count in order to finish execution at the point the programmer is interested in.

However, there are some problems with this method. For one, once the specified number of steps have been executed, the program terminates completely and outputs the current graph. This means that if the programmer wishes to step through the program one rule at a time, for example, they must run the compiler multiple times and increase the number of steps by one each time, wasting time by re-executing earlier steps at each iteration.

Another problem is that, although the compiler was modified, the graphical editor remains unchanged, and so does not support the new partial execution feature. For the feature to be most useful, it would have to be integrated into the editor.

3 Another chapter...

Bibliography

[1] A. Schürr, A. Winter, and A. Zü, "The PROGRES approach: Language and environment," in *Handbook of Graph Grammars and Computing by Graph Transformation*. World Scientific Publishing Co., Inc., 1999, pp. 487–550.

[2] C. Ermel, M. Rudolf, G. Taentzer *et al.*, "The AGG approach: Language and environment," in *Handbook of Graph Grammars and Computing by Graph Transformation*. World Scientific Publishing Co., Inc., 1999, pp. 551–603.

[3] R. Geiß, G. V. Batz, D. Grund, S. Hack, and A. M. Szalkowski, "Grgen: A fast spo-based graph rewriting tool," in *Graph Transformations - ICGT 2006*, 2006, pp. 383–397.

[4] D. Plump, "The design of GP2," in *Proc. International Workshop on Reduction Strategies in Rewriting and Programming (WRS 2011)*, ser. Electronic Proceedings in Theoretical Computer Science, vol. 82, 2012, pp. 1–16.

[5] C. Bak, "GP 2: Efficient implementation of a graph programming language," Ph.D. dissertation, University of York, September 2015.

[6] D. Plump, "The graph programming language GP," in *Algebraic Informatics: Third International Conference, CAI 2009, Thessaloniki, Greece, May 19-22, 2009, Proceedings*, ser. Lecture Notes in Computer Science, vol. 5725, 2009, pp. 99–122.

[7] J. E. Sammet, *Programming Languages: History and Fundamentals*. Prentice-Hall, Inc., Englewood Cliffs, NJ, 1969, ch. 1, p. 18.

[8] M. L. Scott, *Programming Language Pragmatics*, 3rd ed. Morgan Kaufmann, Burlington MA, 2009, ch. 15, p. 806.

[9] Free Software Foundation. (2016, October) GDB: The GNU project debugger. [Online]. Available: https://www.gnu.org/software/gdb/

[10] Oracle. jdb - the Java debugger. [Online]. Available: http://docs.oracle.com/javase/7/docs/technotes/tools/windows/jdb.html

[11] Microsoft. Debugging in Visual Studio. [Online]. Available: https://msdn.microsoft.com/en-us/library/sc65sadd.aspx

[12] Eclipse Foundation. Eclipse IDE for Java developers. [Online]. Available: http://www.eclipse.org/downloads/packages/eclipse-ide-java-developers/neon1a

[13] Free Software Foundation. (2012, November) GDB and reverse debugging. [Online]. Available: https://www.gnu.org/software/gdb/news/reversible.html

[14] P. Wadler, "Functional programming: Why no one uses functional languages," *ACM SIGPLAN Notices*, vol. 33, no. 8, pp. 23–27, 1998.

[15] O. Chitil, C. Runciman, and P. Koopman, "Freja, hat and hood - a comparative evaluation of three systems for tracing and debugging lazy functional programs," in *Implementation of Functional Languages: 12th International Workshop, IFL 2000*, pp. 176–193.

[16] H. Taylor, "Tracing & debugging GP2," Master's thesis, University of York, April 2016.