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], GROOVE [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 is a programming language developed at the University of York [4], an updated implementation of the original language, GP [5]. 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.

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

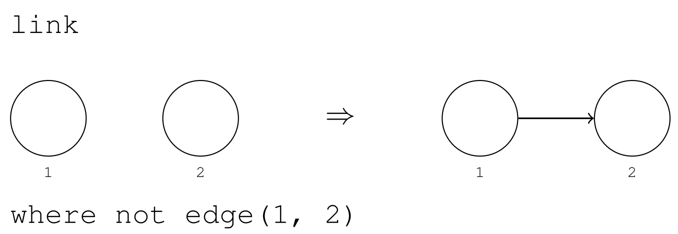

Figure 2.1: A rule in GP2

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. If a match is found, the program moves on to the second phase, the *application*. To ensure consistent output between successive program executions, rule matches are chosen deterministically by the compiled program. If no match is found for the LHS, the rule is considered *failed*.

During the application phase, any nodes and edges present in the LHS but not the RHS will be deleted, and any nodes and edges present in the RHS but not the LHS will be created. At the end of the application phase, the subgraph 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.

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 Subprograms in curly braces "`{P, Q, R}`" define a set, where exactly one subprogram from the set is executed, unless no subprograms in the set can be matched. The subprogram to execute is chosen deterministically.

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 cannot be matched any more. The final attempt to match the LHS will *not* consider the rule *failed*.

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.

An example of a program will go here...

Figure 2.2: Definition of 2-colouring in GP2

An execution of 2-colouring will go here...

Figure 2.3: Example execution of 2-colouring

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 2.2 shows an example GP2 program, the same program used as a case study in Bak's original thesis on GP2 [4, pp.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.

An example execution of the 2-colouring program is shown in Figure 2.3. 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.

2.2 Tracing and Debugging

2.2.1 Debugging in Imperative Languages

When programming in a "classic" imperative language, such as C or Java, it is a given that the programmer will have access to a *debugger*. For C, this may be `gdb` [6], while for Java, it might be `jdb` [7].

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*.

IDEs

Oftentimes, a debugger is 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 [8]. One of the most prevalent Java IDEs, Eclipse, integrates with `jdb` [9].

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.

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.

Reverse Debugging

`gdb` supports what is called "reverse debugging" [10]. 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.

2.2.2 Tracing in Functional Languages

A section discussing what tracing functional languages like Haskell is like. A main focus on the Hat tool for Haskell, describing how it relates to the problem of tracing GP2.

2.2.3 Previous Work on Debugging in GP2

A section discussing the previous project [11] on this topic.

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] A. Rensink, "The GROOVE simulator: A tool for state space generation," in *Applications of Graph Transformations with Industrial Relevance*, ser. Lecture Notes in Computer Science, vol. 3062, 2004, pp. 479–485.

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

[5] 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.

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

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

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

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

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

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