University of York Department of Computer Science

Submitted in part fulfilment for the degree of MEng.

Tracing and Debugging in GP 2

Joshua Asch

2nd May 2017

Supervisor: Detlef Plump

Number of words = 14,715, as counted by `wc -w`.
This includes the body of the report, but not any appendices.

Abstract

The University of York has developed a graph programming language, GP 2, which has its own IDE. In this project, new features are designed, implemented, and evaluated which allow users to step through programs in the IDE in order to reason about them.

Contents

List of Figures

List of Tables

1 Introduction

1.1 Motivation

The University of York has created a graph programming language called GP 2, which has its own Integrated Development Environment (IDE) for editing and running programs using the language.

The current version of the IDE produces "black-box" programs where the user can only see the inputs (the program itself and a graph), and the output (another graph). If the output does not match the user's expectation, either because of a bug in the program or because their expectation was incorrect, they have no way to explore the execution of their program to find out why the program produced that output. The only way to find bugs in a GP 2 program is to reason about the program and try to deduce how the input graph was manipulated to form the output.

This project aims to add additional functionality to the GP 2 toolset which will allow a GP 2 programmer to see their program executed step-by-step in order to determine how exactly the program works. At each step of their program, they should be able to see how the input graph has changed, and what choices and actions the program is taking. This will allow programmers to reason in-depth about their programs to make it easier to find and fix bugs, or to strengthen their understanding of GP 2 as a language.

1.2 Ethics

This project is not related to defence and is not safety-critical, so the impact of the work is very low from an ethical standpoint. No humans or animals are involved in the development of this work.

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. When using a graph programming language, the problem being solved is 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 "classic" 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 GROOVE [1], AGG [2], GrGen [3], and, most recently, GP 2 [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 a number of languages, including Java (in the case of AGG and GROOVE), C# .NET (GrGen), or C (GP 2). This program can then be executed to find the output graph generated by the algorithm.

2.1.2 The GP 2 Language

GP 2 (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 graph programs at a high level, performing graph transformations without having to implement data structures to represent the graphs in more traditional lower level languages.

Programming in GP 2 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 GP 2 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 GP 2 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 GP 2 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*,

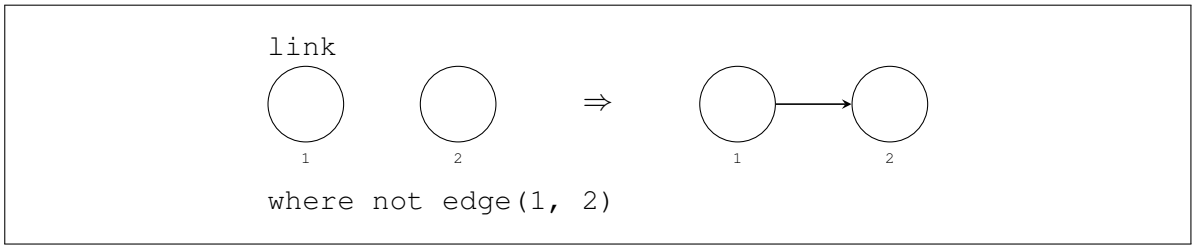

Figure 2.1: A rule in GP 2

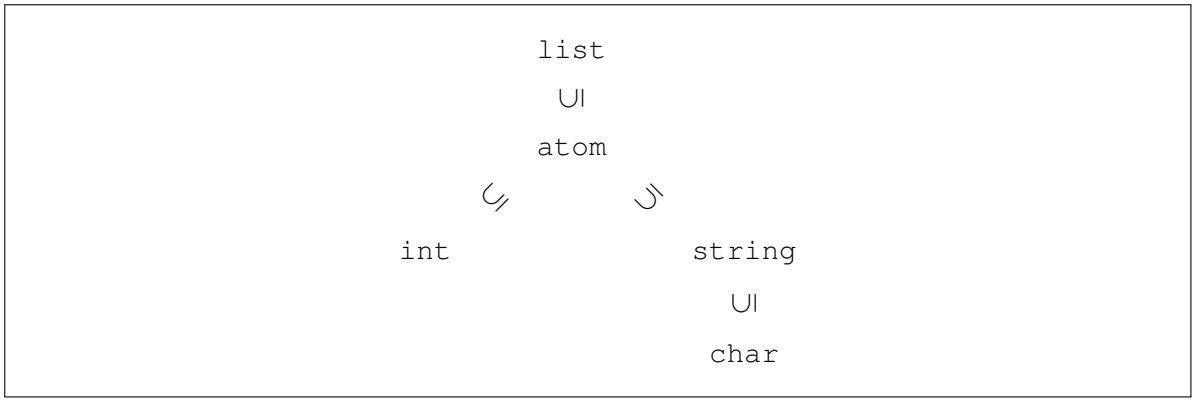

Figure 2.2: GP 2's type hierarchy for labels

meaning that every item in the LHS must be matched with exactly one item in the host graph.

In this implementation of GP 2, rule matches are chosen determinstically due to the impracticality of generating all non-deterministic possibilities, which is how GP 1 worked. 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 indicatedby the small numeric identifers below them. 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 will be an injective match to 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.

Labels

Nodes and edges in a graph defined in GP 2 can also have *labels*. Labels can be used cosmetically, without affecting the execution of the program, or they can be included in rules to have the program read and modify labels.

GP 2 has a type hierarchy for labels, shown in Figure 2.2. Labels can be of any of these five types. An `atom` consists of a single entity, either an `int` or a `string`. A `list` is a collection

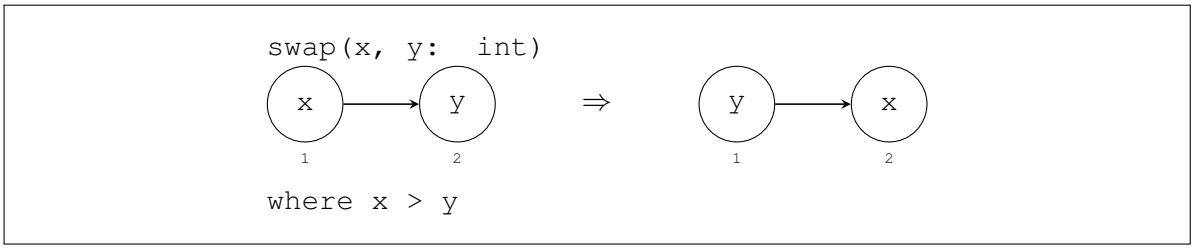

Figure 2.3: A GP 2 rule which uses labels

of zero or more `atom`s concatenated together; the concatenation is shown using a colon (`:`).

Host graphs and the LHS and RHS of rules must be *totally labelled*, meaning that every node and edge must have a label. If the programmer wishes to use a node or edge which appears to be unlabelled, they can use the empty list for that label.

Labels behave in two subtly different ways depending on where they are used. In a graph, a label has a specific value, similar to an integer or string literal in a convential programming language. When used in a rule, labels can either use literal values to match nodes with exact label values, or they can use expressions which can match a range of label values.

When matching a rule involving labels, GP 2 finds a variable assignment which matches the expressions of the labels in the graph. These same values are then used in the RHS of the rule, if the same variables are referenced.

When using labels in a rule, the type of each label must be specified so that GP 2 can find an assignment for each one. A rule which specifies a certain label type can match any label with that type or a more specific type in the type hierarchy. For example, a rule which contains a label of type `string` can match a node which has a `char` label. The `list` type will match any label.

The rule in Figure 2.3 will only match a pair of nodes where the labels are both integers, and the value of node 1's label is greater than the value of node 2's label. This rule demonstrates that labels can be used as conditions in the program, and also that a rule can modify labels. It also shows that labels act like variables when used in a rule, since the values of `x` and `y` are re-used in the RHS.

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 2.4 shows the result of attempting to apply a rule where the match violates the dangling condition. Since node 2 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 4 to 2 should be deleted. This leaves a *dangling edge* from node 4 with no target. The dangling condition is used to ensure that rule applications result in no dangling edges.

Programs

A GP 2 program defines the order in which to apply rules using 9 simple control structures:

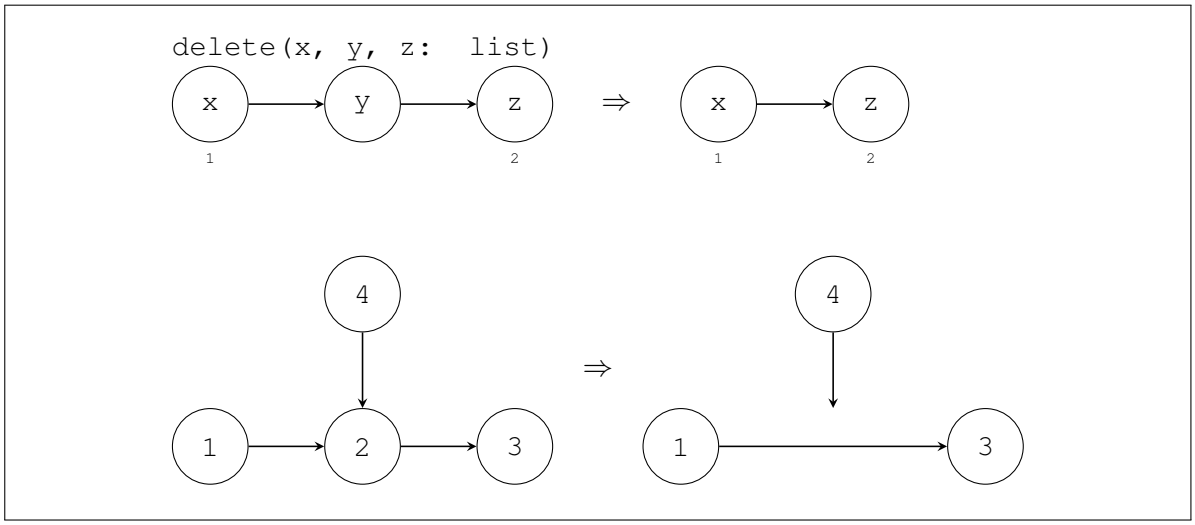

Figure 2.4: Attempting to apply a rule when the match violates the dangling condition

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, chosen non-determinstically from all rules in the set which have a valid match. If there are no rules in the set which match, the program fails in the same way as if a single rule failed to match.

IF-THEN-ELSE In the statement "`if C then P else Q`", a copy of the graph is taken, before executing the sub-program `C`. The result, i.e. success or failure, is then used to determine whether to run `P` or `Q`. If C succeeded, `P` is executed on the copy of 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 always 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 / LOOP A subprogram followed by an exclamation point `P!` is executed repeatedly until it fails. When `P` fails, the graph is reverted back to the last successful state, that is, the state it was in at the end of the last successfully applied rule from `P`.

The keyword `break` can also be used to exit from a loop before reaching the end. When using `break`, the graph is *not* reverted before continuing with the program.

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. Procedures are not allowed to be recursive.

SKIP A no-op which always succeeds, and does not affect the graph, which is invoked using the keyword `skip`. This is the same as executing a rule with an empty LHS and an empty RHS, which always matches and makes no changes to the graph.

Fail A statement 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`. It can also be simulated using an empty Rule Set `{}`.

Or In the statement "`P or Q`", exactly one subprogram, either `P` or `Q` is chosen non-deterministically and executed. Unlike the Rule Set construction, if the chosen subprogram fails, the program does not attempt to execute the other, and the program fails.

In GP 2, a subprogram is either a single rule, referenced by its name, or one of the above control structures. It is therefore 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 2.5 shows an example GP 2 program, an adapted version of the program used as a case study in Bak's thesis on GP 2 [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 requires that the graph is *connected*, meaning there is an undirected path between every pair of nodes.

The 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 GP 2, 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 nodes are coloured red.

An example execution of the 2-colouring program is shown in Figure 2.6. 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 the loop 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 `fail` statement from theIf-Then-Else. This in turn causes the outer `try` to fail, reverting all changes made to the graph and returning the uncoloured input graph as the program's output.

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 coloured graph is returned as the result of the program.

In the example execution in Figure 2.6, 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

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

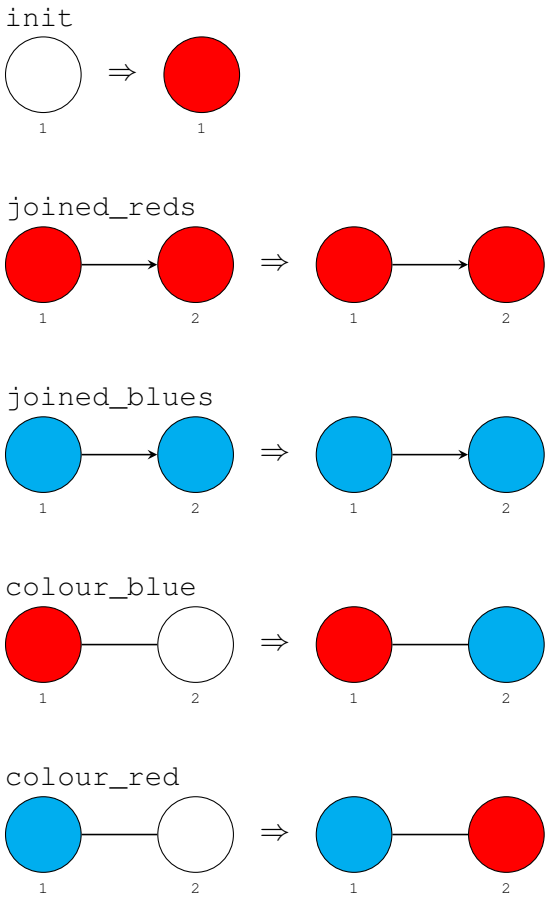

Figure 2.5: Definition of 2-colouring in GP 2

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

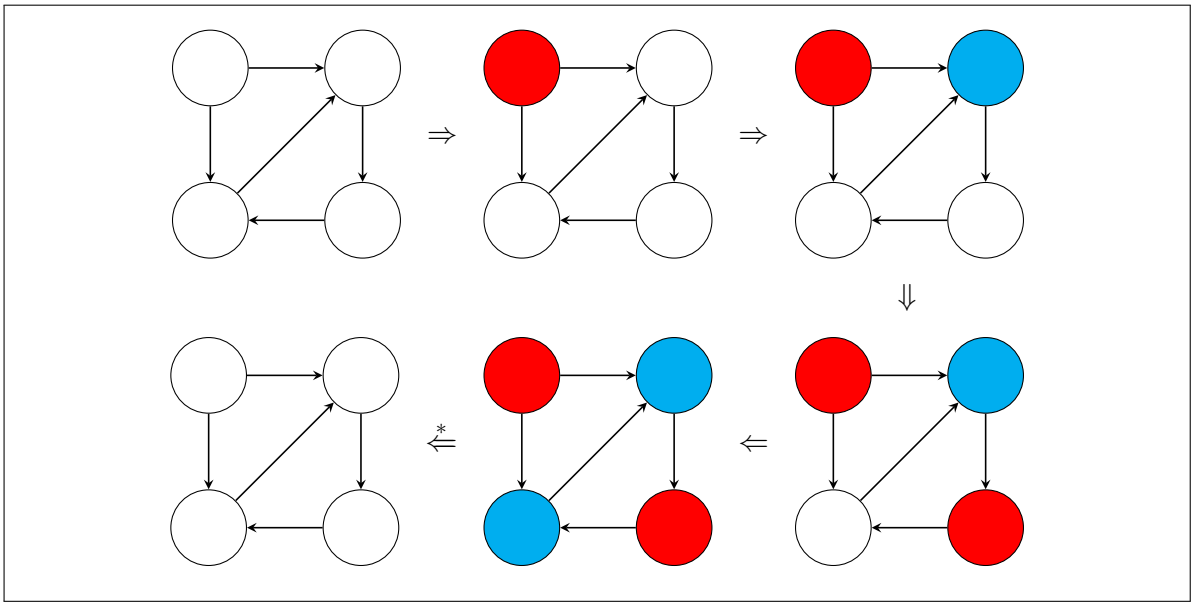

Figure 2.6: Example execution of 2-colouring

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 GP 2 has a rule-based structure, unlike imperative programming languages, it is not clear what a "line of code" represents in a GP 2 program. The textual part of a GP 2 program can indeed have lines, but since a single line in a GP 2 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 GP 2, 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 GP 2.

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 GP 2. For instance, when stepping through a GP 2 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 GP 2. 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, it is not clear how this feature could be implemented in GP 2. It would not make sense for the user to be able to edit the input graph or an intermediate graph while execution is paused, because the result of the program would be incorrect. Editing the rules and program text would make more sense, but this could lead to problems, such as if the user pauses execution after matching a rule, and then edits that rule in a way that makes the match incorrect.

Reverse Debugging

`gdb` supports what is called *reverse debugging* [13]. This allows program execution to 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 GP 2, because it would allow a programmer to step backwards through the execution of a GP 2 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 GP 2, because it would only need to trace rule matches and applications, both of which are implemented inside GP 2 itself.

2.2.2 Tracing in Functional Languages

The syntax and execution of a GP 2 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 GP 2; 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 GP 2 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 available. For Haskell, for instance, a handful of different tracing tools are available [15]. Two of them, Freja and Hat, are modified Haskell compilers which automatically add 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.

Another disadvantage of Hat is that it has a very large impact on performance, running more than ten times slower than the original Haskell program [15]. Hat adds a lot of overhead at runtime, since the program has to do a lot of extra work to store the tracing data. On the other hand, Freja is also a tracing tool, yet Chitil, Runciman, and Koopman state that the overhead is "not noticeable". It is clear that the performance impact of these tools depends

on their implementation.

2.2.3 Debugging With Other Graph Programming Tools

Each of the three other graph programming tools mentioned in Section 2.1.1 have their own debugging or tracing tools.

AGG

Of the three tools, AGG has the simplest debugging features. It allows the user to select a rule in their graph grammar and then look through all the matches for that rule, by showing them on the current graph.

A unique feature of AGG is that it also allows the user to manually create a match; by selecting nodes and edges first in the LHS of the rule, and then in the graph, the user creates a mapping between the rule and the current graph to create a match. The tool will alert the user if they attempt to select a mapping which is not a true match.

When a match has been chosen, either automatically or manually, the rule can then be applied by clicking a `Transformation Step` button in the editor. After a rule has been applied, an `Undo Step` button becomes available which reverts the changes made by the most recent rule application.

GROOVE

GROOVE also allows the user to inspect all possible rule matches for the current graph. In the list of rules shown in the editor, the possible matches for each are shown underneath each rule. These matches can be clicked to highlight them on the graph, and double-clicking applies the rule, updating the graph.

Similar to AGG, it is possible to undo a single rule application. However, GROOVE also keeps a history of all previous graph states. In the "State Space Explorer", the tool shows a transition diagram containing all the previously visited graph states, with the transitions representing the rule applied to reach a given state. In this view, the user can select a previous state and jump back to it, allowing the user to undo multiple rule applications at once, or even undoing previous rule applications and re-applying other rules, in one step.

GrGen

GrGen's debugger is different in that it is controlled using a command-line style interface, rather than using buttons in the editor UI. It provides a command to apply a single rule, similar to the other two tools.

However, when applying a rule, GrGen can show more detail. It can show the steps required to apply a rule: the match, the modification to the graph, and the result. It shows each of these stages individually, highlighting the relevant areas on the graph at each stage.

Another unique feature of GrGen's debugger is that it allows the user to tell the program to run until the end of the current loop. GrGen supports a loop structure similar to GP 2's As-Long-As-Possible, and the debugger's `step out` command will continue to execute until the current loop ends, before giving control back to the user.

Finally, GrGen also supports a feature similar to breakpoints. It allows the user to specify a condition, and when that condition is met, execution will pause. One type of condition

which can be specified is a certain modification to the graph, such as pausing execution when a specific node is deleted from the graph.

2.2.4 Previous Work on Debugging in GP 2

There has been some previous work to add debugging facilities to GP 2 [16]. This work was focused on modifying the compiler to support stepping through a program.

In that project, the GP 2 compiler was modified to add command-line arguments to compiled GP 2 programs, which allow the user to specify how big a "step" should be, and how many steps to execute before stopping. This allows the user 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 program 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 Requirements

Before designing and implementing a system, the requirements for the system need to be considered. These requirements should be based on expected real-world usage of the tool and keep the goals of the tool in mind.

3.1 Potential Users

In order to decide what the requirements for a debugging tool are, it is useful to consider the types of user who would potentially use such a tool. There are two main potential audiences for a GP 2 debugging tool:

- Experienced GP 2 programmers who have written a GP 2 program which they would like to debug. Either the program produces unexpected output, or it never terminates, so the programmer wishes to use the debugging tools to understand why their program is behaving incorrectly, and then fix the bug.
- New users of GP 2 who are still learning the language. GP 2 is an unconvential language and its concepts can be unfamiliar to inexperienced users. Allowing these users to step through a GP 2 program would let them see how GP 2 works and help them to understand the language faster.

  However, it should be noted that the debugger is not a substitute for documentation or user manuals; the debugger should not be expected to explain the semantics of GP 2, but rather act as a tool to show the process of running a GP 2 program.

3.2 Integration and Maintainability

Consideration should also be given to how new features are integrated into GP 2. The experience of using existing GP 2 features should be unaffected, and the codebase should stay maintainable for future developers of GP 2. The high-level requirements this leads to are:

- Backwards compatibility. Adding a debugging tool should not impact the existing features of GP 2, the compiler, or the IDE. Programs written before the addition of debugging tools should still be runnable in the new IDE.
- Make debugging optional. The user should not be forced to use the debugging tools every time they run their program, and should be able to choose when to enable them.
- Minimise the performance impact. Running a program with debugging tools should affect performance as little as possible, and running with debugging disabled should have no performance impact compared to the previous version of GP 2.
- Keep the codebase maintainable. When modifying existing code, or adding new code, follow conventions and make code readable. Make the intent of any changes clear to ensure future developers will understand what has been changed.

```
Main = if (init; Colour!; Invalid)
           then add_red_node
           else add_green_node
Colour = {colour_blue, colour_red}
Invalid = {joined_blues, joined_blues}
```

Figure 3.1: Alternate version of the 2-colouring program

- Consider dependencies between components. Implementing debugging will require modifying both the compiler and the IDE. It is important to consider the dependencies between these components, to ensure that they can still function independently.

  It is also important to consider the dependencies between new code and existing code. If pre-existing bugs are discovered in the compiler, for example, it must be possible to fix them without needing to modify the code added for debugging, and vice versa, since compilation and debugging should be separate components. Compilation should still work independently of debugging.

3.3 User Scenarios

To further identify specific low level requirements for the tool, two hypothetical user scenarios have been created to show potential situations a user could be in, and how the debugging tools can help the user complete their task in those situations.

3.3.1 Experienced GP 2 Programmer

In the first scenario, the user is an experienced GP 2 programmer who understands all the concepts and semantics of GP 2. They have written a program similar to the one shown in Figure 2.5, to determine whether a graph is two-colourable.

However, their program is slightly different, and does not output the coloured graph when it is two-colourable; instead, the original, uncoloured graph is always returned, and an extra node is added to show the result. If the graph is two-colourable, a green node is added, otherwise a red node is added. The program they have written is shown in Figure 3.1, along with the two new rules, `add_red_node` and `add_green_node`. The other rules are omitted since they are identical to the rules in the original program.

When they test their program, for some inputs it gives the incorrect result; for certain non-two-colourable graphs, the program still adds a green node to indicate that it is two-colourable. Just by inspecting the program, the user cannot determine what the bug is with the program, so they decide to use the simplest possible non-two-colourable input graph, shown in Figure 3.2, and use the debugger to step through their program.

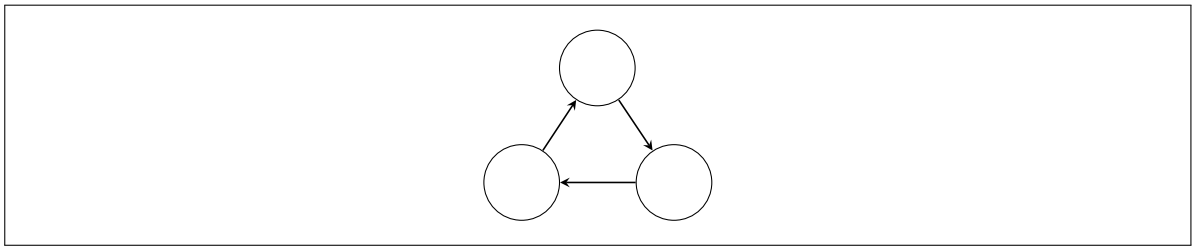

Figure 3.2: Simplest possible non-two-colourable graph

As they step through the program, the `init` rule colours the top node red, then `colour_blue` is applied, followed by `colour_red`. This results in the graph being coloured red, blue, red running clockwise. Clearly the rules for colouring nodes are defined correctly, so the bug must be in the error detection logic.

They continue stepping through the program and it reaches the `Invalid` procedure. They use the debugger to highlight a match for `Invalid` without applying it, and it turns out there are none, which should not be the case. They first look at the definitions for the `joined_reds` and `joined_blues` rules, and they are definitely correct. Finally they look at the definition of `Invalid` itself, and realise they have put `joined_blues` in the rule set twice, so `joined_reds` is never actually used.

The bug in this program was fairly subtle, since the general structure of the program is mostly correct; there are two rules in `Invalid` as expected, but the same rule was mistakenly listed twice. Without the debugger to step through the program, it may have taken much longer to spot this mistake, since stepping through allowed the user to see the coloured graph before it was reverted by the `if` statement.

The ability to view a match without applying it also helped in this case, since it allowed the user to see that none of the rules in `Invalid` matched, without automatically exiting the `if` condition when no rules matched.

3.3.2 New User Learning GP 2

In the second scenario, the user is inexperienced with GP 2 and is still trying to learn the language. They understand most of how the language works, but they are having trouble understanding the semantics of If-Then-Else.

They have written a very simple GP 2 program, shown in Figure 3.3, which consists of three rules and a single If-Then-Else statement. When they run this program on the input graph shown, they incorrectly expect the output graph to match the RHS of the `bridge` rule, because they expect `insert` to be applied, followed by `bridge`.

In actuality, this program fails, because the changes made by `insert` are reverted before attempting to match `bridge`, meaning there is no possible match for `bridge`.

The result confuses the user, who then decides to use the debugger to step through the program to see what happened. The first thing they see is the program tries to match `insert` and succeeds, which they expected. However, then they advance to the next execution step and are surprised that the changes to the graph have been reverted. They then realise that they had mixed up the semantics of If-Then-Else and Try-Then-Else.

Ideally in this situation, the debugger would explicitly show that backtracking will occur once the `if` condition has been verified. It may also allow the user to step backwards and

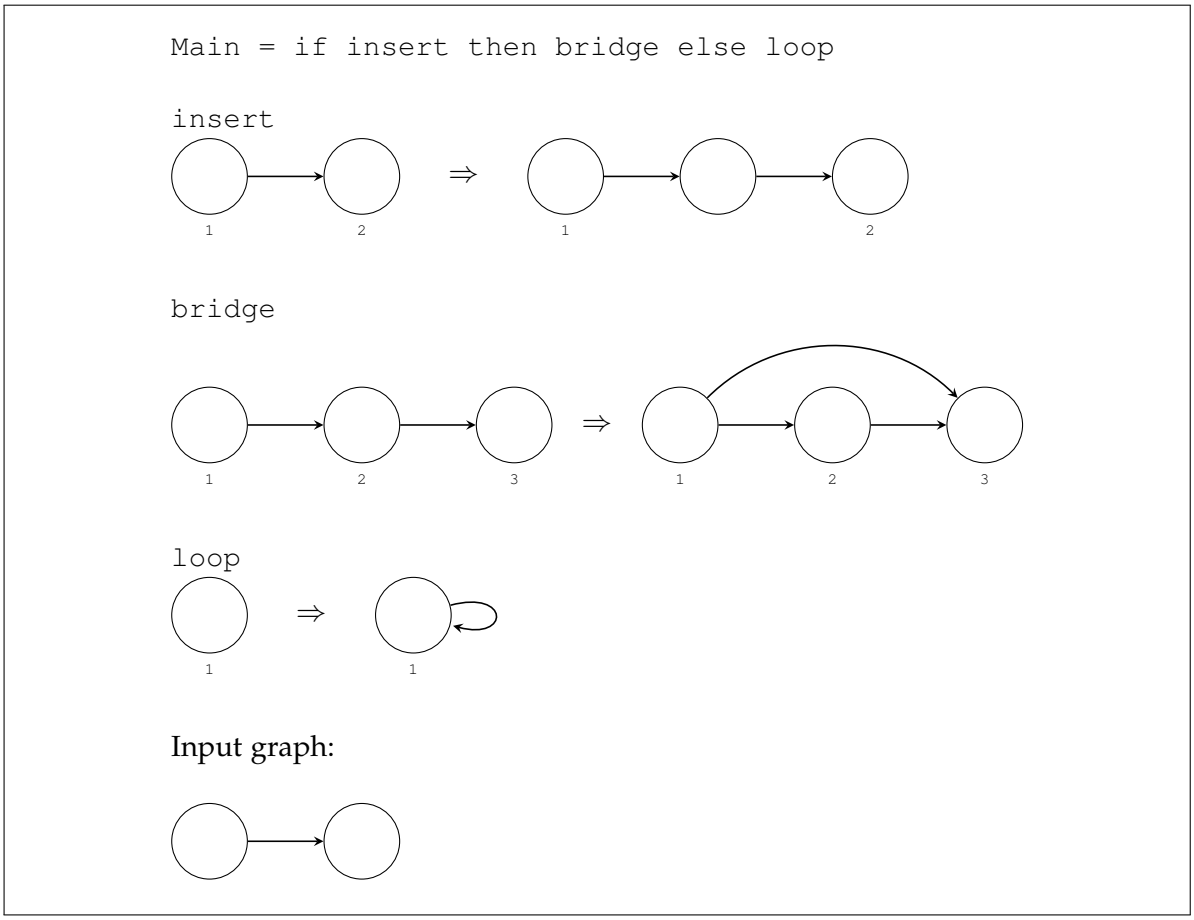

Figure 3.3: Simple program to learn the semantics of If-Then-Else

forwards over the program so they can make sure they understand what is happening.

By inspecting the execution of this simple program, the user has successfully used the debugger to aid their learning; even though the debugger does not explain the semantics directly, the user can use it to understand how the program works.

3.4 List of Requirements

Table 3.1 shows a list of all requirements which have been identified for the GP 2 debugging tool.

ID	Description
1	The system must be able to show execution of a GP 2 program step-by-step.
1.1	The system must be able to apply rules one at a time to step through a program.
1.2	It is desirable for the system to allow the user to step backwards through a program.
1.3	It is desirable for the system to be able to execute an entire control sequence in one step.
2	The system must have a visual debugging component.
2.1	The system must show the rule being applied when stepping through a program.
2.2	The system must highlight the next rule match in the graph.
2.3	The system must highlight the current position in the textual portion of the program.
2.4	It is desirable for the system to highlight changes which were made as a result of the previously applied rule.
2.5	The system must show each intermediate graph in the IDE.
3	The system must allow debugging to be controlled from within the GP 2 IDE.
3.1	The system must provide a way to advance execution by one step.
3.2	It is desirable for the system to provide a way to backtrack execution by one step.
3.3	The system must provide a way to find a match for the next rule, but not apply the rule. (See requirement 2.2)
3.4	The system must provide a way to apply a rule for which a match has been found but not yet applied. (See requirement 3.3)
3.5	The system must provide a way to run an entire program without debugging it.
4	The system must act in a predictable and consistent way during each execution of a program.
4.1	The system must produce identical results whether debugging is enabled or not.
4.2	The system must be able to load existing projects created before the system was modified.

Table 3.1: List of system requirements identified for the debugging tool

4 Design

4.1 Design Considerations

4.1.1 Implementations of GP 2

There are two implementations of GP 2, an interpreter written in Haskell [5, 17], and a compiler written in C [5, 18]. The interpreter is intended to provide a reference for other implementations of GP 2, and for the GP 2 developers. The GP 2 IDE uses the compiler, rather than the interpreter, to run the user's program.

The interpreter may be of some interest, since an interpreter runs "live", moving through the program executing each statement as it is reached. This behaviour could be exploited to implement a form of debugging, by pausing execution of the interpreter at various points and outputting the current state of the program.

To take advantage of the interpreter, the GP 2 editor would have to be modified so that it runs the user's program using the interpreter rather than the compiler.

The other difference between the interpreter and the compiler is that the compiled program only produces one output for a given input to a program. The interpreter returns all unique output graphs, up to isomorphism. For debugging purposes, this may be of interest to the user, since it allows them to see all the possibilities of their program. To the developer of the debugger, this is a disadvantage, since the GP 2 IDE does not currently support displaying all possible outputs, so this functionality would require additional work.

Additionally, because the interpreter is only a reference implementation, the compiler is the implementation intended for most users. If the IDE were to be changed to run the interpreter for debugging programs, checks would have to be made to ensure that the interpreter produces identical outputs to a compiled version of the same program. This would involve using the interpreter in single-result mode, which causes the interpreter to return one result graph instead of all possible results. However, it still cannot be guaranteed that the single result will match the result produced by the compiler.

4.1.2 Backtracking in the GP 2 Compiler

To support some operations, such as If-Then-Else, the GP 2 compiler uses *backtracking*, where any changes to the graph are reversed [5, ch. 5.8]. At the end of the conditional part of an If-Then-Else, any changes need to be reverted before continuing.

This feature of the compiler could be used as part of the debugging features, by using it to allow the user to step backwards through the program as it runs, similar to the reverse debugging mentioned in Section 2.2.1. It would allow the user to see the state of the graph before a certain rule was applied, for instance.

However, this backtracking is not available at all points in a program's execution; backtracking code is only added by the compiler if static analysis shows that it is necessary. For example, the compiler does not generate backtracking code for rules outside of conditionals, since these changes are never reverted. The compiler would have to be modified to add backtracking code to the entire program.

Communicating Backtracking to the User

When the user is debugging a program which causes backtracking to occur, the tool must indicate in some way that this has happened. For inexperienced users especially, if the graph reverts back to a previous state with no explanation, it could be extremely confusing.

There are multiple ways the debugging tool could communicate backtracking to the user:

- Show a message when backtracking occurs. This is the simplest option, which would involve showing a message somewhere in the IDE when the graph is reverted. This would explain to the user what has happened, although they may still not understand why the backtracking occurred, or what previous state was restored.
- Highlight the part of the user's program where the backtracking is rooted. For instance, if the program contains "`if r1 then p1 else p2`" (assuming `r1` is a rule and `p1` and `p2` are procedures), "`if r1`" could be highlighted differently to indicate that it may produce backtracking. This would give the user more insight into why the backtracking has occurred, although it may still not be clear which previous graph state has been restored.
- Show the previous state somewhere in the IDE. This method would provide the most context. It allows the user to see exactly what will happen when backtracking occurs. It could also be combined with the previous idea to give additional information. Combining both would allow the user to see the cause of the backtracking, and also what changes occurred.

4.1.3 Non-determinism

While the definition of GP 2 includes non-determinism, such as when choosing a rule from a Rule Set, the implementation of the C compiler is mostly deterministic. When executing a Rule Set, the generated C program will always select the first rule in the set which matches, rather than non-deterministically selecting from all the rules which match. This is shown in Listing 4.1, which shows the code which the compiler generates for the statement "`{r1, r2, r3}`". This code tries each rule in turn, and stops when it finds the first one which has a match.

Listing 4.1: Generated code for a Rule Set structure

```
do
{
  if(matchMain_r1(M_Main_r1))
  {
    applyMain_r1(M_Main_r1, false);
    success = true;
    break;
  }
  if(matchMain_r2(M_Main_r2))
  {
    applyMain_r2(M_Main_r2, false);
    success = true;
    break;
```

```
  }
  if(matchMain_r3(M_Main_r3))
  {
    applyMain_r3(M_Main_r3, false);
    success = true;
  }
  else
  {
    ...
    // Failure handling omitted for brevity
  }
} while(false);
```

If a user is familiar with the GP 2 specification, they may expect their program to behave non-deterministically, such as selecting rule matches non-deterministically, or producing multiple possible output graphs. They may also expect features in the debugger related to this non-determinism, such as showing all the possible matches for a rule, or showing all the matching rules in a Rule Set. However, because the compiler produces mostly deterministic programs, it is not currently possible to support anything like this in the debugger.

For example, the generated code for a Rule Set does not attempt to match any other rules once a match is found, so the debugger could not show all matching rules. Enabling this would require modifying the compiler to find all matches before picking one, even if it still always picked the first match.

The exception to this is the Or statement. In the GP 2 compiler, this statement does not always choose the same option, but instead uses the `rand()` function to randomly choose between the two possibilities. This seems to act counter to requirement 4 for the debugger, which states that the system must be predictable, because on each execution of the program, it is random whether the first option or second option will be chosen.

However, the debugger cannot take responsibility for how the user's program is structured. Listing 4.2 shows plain C code which is analagous to a GP 2 Or statement, randomly calling one of two functions based on a random number. A C programmer would not expect the debugger to always follow the same execution path when debugging this code, so it is reasonable to take the same approach with a GP 2 debugger.

It would also not be possible to show the user the different outcomes which would occur in each branch of the Or statement, because only one is executed at runtime, and also because compiled GP 2 programs are only capable of producing one output graph. It would require a lot of re-implementation in the compiler to support multiple outputs.

Listing 4.2: C code analagous to GP 2's Or statement

```
int random = rand();
if ((random % 2) == 0) {
    someFunction();
}
else {
    someOtherFunction();
}
```

In a similar vein, the order of rules within a Rule Set can affect the execution of a program,

since the first matching rule is always applied, as discussed previously. This means that if the user edits their program and re-orders the rules in a Rule Set, their program may give different results. Again, this issue is due to the structure of the program, which the debugger cannot take responsibility for.

4.1.4 Program Representation

When a user creates a graph using the visual editor in the IDE, the graph must be converted to a text-based format in order to compile and run the GP 2 program. Depending on the position of elements in the editor, isomorphic graphs may be converted to different textual representations, with the nodes and edges listed in a different order, for example.

This is relevant to the debugger because when running a program, the order of nodes and edges will affect which matches are selected when applying a rule. If the user runs their program and then rearranges the nodes in the editor, running the program again may give a different result graph. This would be surprising to a user, since they did not modify the structure of the graph, only its layout.

Without modifying the algorithm for converting a graph to the text based representation, it is not possible to avoid this problem. It may be useful to warn the user if the representation of their graph changes, so that they are aware it may give different results.

4.2 Proposed Solutions

Two different possible high-level designs are outlined below, along with the advantages and disadvantages of each.

4.2.1 Modify Compiler to Run Partial Programs

Following a similar method to the previous work on adding debugging to GP 2 [16], the compiler would be modified so that part of a program can be run before stopping and giving the intermediate result. The IDE would interact with the compiler to control how much of the program to run, and to display the intermediate graphs to the user.

The IDE would have two modes for running a program. The first mode would act exactly as the current version of the IDE, running the entire program and displaying the result. No changes would need to be made to the IDE for this mode.

The second mode would be for debugging the program, and would provide the user with controls to step through their program. Each time a step is taken, the compiler would be invoked on the most recent intermediate graph, solving the problem from the previous project of having to re-run the first steps. To enable this, the output must also include some reference to the point which has been reached in the program, in order to continue from the same point next time.

Advantages

- Similar work to this has already been done [16], so implementing this design would be an extension of that rather than a new solution created from scratch.

- Non-terminating or long-running programs do not have to be run in their entirety, saving the user time when debugging a program, since they do not have to wait for execution to finish.
- As shown in the previous project, it is possible to implement partial execution in the compiler without affecting compilation or runtime performance. It is likely that the tracing method would impact runtime performance in some way.

Disadvantages

- It would be difficult to allow the user to step backwards through the program with this method, because each invocation of the compiler would be separate, meaning the use of backtracking would be impossible, and if the output is simply a graph and the current position in the program, there is no way to discover what changes were made to the graph during the last invocation.
- Continuing from a partially executed state would be impossible if the user edits their program or the input graph before continuing. While this would be an error on the user's part, it could potentially be confusing if they edit the program to fix a bug and then try to continue from the previous point.
- Adding various granularity options for partial execution would require adding many new command-line arguments to the compiler. Having a lot of command-line arguments related to debugging may confuse users, and could also lead to potential bugs when using these arguments programmatically from the IDE.

4.2.2 Use Tracing to Step Through Execution History

Instead of pausing execution to show the current state of the program, this method would instead add tracing to the user's program, and the IDE would run the entire program before stepping through the history. This is similar to how tools such as Hat (discussed in Section 2.2.2) work.

The existing way of running a program in the IDE would remain unchanged, and would not enable tracing, to avoid any performance impact of having tracing turned on. A second button in the UI would run the program with tracing turned on, and then switch the IDE into a mode which allows the user to examine the execution history of their program.

Advantages

- It would be possible to add the ability to jump to any point in the program's execution, since all states would be stored. Using the other proposed method, jumping to a specific point would involve running the program and somehow detecting when the requested state is reached.
- Stepping through a program would be much faster than using partial execution because another process does not have to be executed at each step.
- Debugging would not be dependent on the compiler, once the program has been compiled and run, so certain additional debugging features could be added to the IDE without requiring the compiler to be updated.

Disadvantages

- Debugging a non-terminating program may be impossible because the program may never output a trace, unless the compiler can be modified to produce programs which terminate upon receiving a signal, and save their trace before exiting.
- Storing a trace may use a lot of memory or disk space, depending on the implementation.
- It would not be possible to add features such as choosing a rule to apply, or a match to apply a rule to, since execution has already finished when the user regains control.

4.3 Chosen Solution

The second solution, tracing a program and then allowing the user to step through the execution history, has been selected as the chosen design to implement debugging features in GP 2.

This solution has advantages over the other design, such as the ability to more easily implement more program traversal methods, including backwards stepping and jumping to a specific point in time. Each step through the trace will also be much faster that the other design; the IDE does not have to wait for the program to run a step and return the results, because the program has already been run.

Another advantage of this design is that it can also help the problem of debugging a program which uses the Or statement. Since the trace can be used repeatedly once the program has finished running, the user can step through the program multiple times without having to re-run the program. This means that the branch chosen when stepping through the Or statement will remain constant until the user edits the program and has to run it again.

The solution also has trade-offs, such as a possible impact on runtime performance; extra work needs to be done to store the trace. Runtime performance, while one of the requirements outlined in Section 3.4, is less important than the user's ability to actually debug their program. For the same reason that IDEs such as Visual Studio disable compiler optimisation by default when debugging, an accurate debugger is more useful than a higher performance one.

A lot of memory and disk space is available on modern hardware; it is unlikely that a GP 2 trace would exhaust the available resources on a relatively recent machine. For example, in [17], a program is given to generate a Sierpinski triangle, a task which is known to be difficult and was posed as a competition for graph transformation tools [19]. Experimentally, running this GP 2 program using the current version of the GP 2 compiler produces a C program which can produce the 12th-generation Sierpinski triangle, containing over 797,000 nodes and 1,594,000 edges, while using less than 250 MB of memory at runtime.

Additionally, a trace format will need to be designed and implemented so that the program and the IDE can communicate. This will be a simple Domain-Specific Language (DSL) which represents the actions a GP 2 program takes. Part of this format can be based on the existing DSL which defines the machine readable format of graphs and programs in GP 2.

Finally, while the other proposed design would have trivially allowed non-terminating programs to be debugged, this design does not have a clear solution to this problem. It is more difficult to support non-terminating programs with tracing, since at first glance it appears the program has to finish running before it can output a trace. However, if the file

is written while the program is still running, rather than storing the trace in memory and writing it to disk at the end, a partial trace can still be retrieved even if the running program is forcefully terminated by the user.

4.3.1 Trace File Format

The running program and the IDE must communicate in some way in order for the debugging interface in the IDE to show the execution of the program. To do this, the program trace is stored in a file on disk, so that the program can write to it and the IDE can read from it.

The format used for these files is completely internal to the GP 2 system and does not need to be edited or read directly by users. This means that the format can focus on requirements such as being easy to parse in the IDE's C++ code, rather than focusing on making it human readable.

Two formats were considered for trace files, one formatted as an XML document, and one formatted as a plain text document with a custom structure inside. An example trace file in each format is shown in Listing 4.3 and Listing 4.4. These examples both show a theoretical execution of the incorrect 2-colouring program from Figure 3.1.

Listing 4.3: XML based trace file example

```
<?xml version="1.0" encoding="UTF-8" ?>
<trace program="2_colouring.gp2" input_graph="triangle.host">
  <if backtracking="true">
    <condition>
      <rule name="init">
        <match>
          <node id="0" />
        </match>
        <apply>
          <remarkNode id="0" old="0" new="1" />
        </apply>
      </rule>

      <loop>
        <iteration>
          <procedure name="Colour">
            <ruleset rules="colour_red,colour_blue">
              <rule name="colour_blue">
                <match>
                  <node id="0" />
                  <node id="1" />
                  <edge id="0" />
                </match>
                <apply>
                  <remarkNode id="1" old="0" new="2" />
                </apply>
              </rule>
            </ruleset>
          </procedure>
```

```
        </iteration>

        <iteration>
          <prcoedure name="Colour">
            <ruleset rules="colour_red,colour_blue">
              <rule name="colour_red">
                <match>
                  <node id="1" />
                  <node id="2" />
                  <edge id="1" />
                </match>
                <apply>
                  <remarkNode id="2" old="0" new="1" />
                </apply>
              </rule>
            </ruleset>
          </prcoedure>
        </iteration>
      </loop>

      <procedure name="Invalid">
        <ruleset rules="joined_blues,joined_blues">
          <rule name="joined_blues">
          </rule>
        </ruleset>
      </procedure>
    </condition>

    <else>
      <rule name="add_green_node">
        <match></match>
        <apply>
            <addNode id="3" label="empty" mark="3" root="false" />
        </apply>
      </rule>
    </else>
  </if>
</trace>
```

Listing 4.4: Plain text trace file example

```
PROG ; 2-colouring.gp2 ; triangle.host
IFFS ; 1
CNDS
RULE ; init ; n0
MARK ; n0 ; 0 ; 1
LOPS
ITRS
PRCS ; Colour
SETS ; colour_red,colour_blue
RULE ; colour_blue ; n0,n1,e0
MARK ; n1 ; 0 ; 2
SETE
PRCE
ITRE
ITRS
PRCS ; Colour
SETS ; colour_red,colour_blue
RULE ; colour_red ; n1,n2,e1
MARK ; n2 ; 0 ; 1
SETE
PRCE
ITRE
LOPE
PRCS ; Invalid
SETS ; joined_blues,joined_blues
RULE ; joined_blues ; fail
SETE
PRCE
CNDE
ELSS
RULE ; add_green_node ; ()
NEWN ; 3 ; () ; 3 ; 0
ELSE
IFFE
```

In the XML format, each action in a program is represented by a tag. The attributes are used to provide details, such as the label of a node, or whether an If-Then-Else required backtracking.

In the plain text format, each action is represented by a line of text similar to a line in assembly; it begins with a four-letter code representing the operation, which is followed by the details for the operation, separated by semicolons.

The XML format was ultimately chosen because it has multiple advantages over the plain text format.

One advantage of the XML format is that the Qt framework, which the GP 2 IDE uses, has a built in XML parser which would make implementation much easier. Using the plain text format would require a custom parser to implement. GP 2 project files (.gpp files) also

use XML, meaning it can be expected that other GP 2 developers understand how to use the XML parser from the Qt framework, making the code easier to maintain.

In addition, the XML format is easier to understand by a user if they open the file directly. This is because each of the parameters are named, since XML tag attributes are name-value pairs. In the plain text format, the parameters are simply listed with no other information, making it harder to determine what each one means. The XML tag names are also more descriptive than the four letter codes from the plain text format.

Whilst readability is not an important factor to end users, it increases maintainability for other GP 2 developers, because it is easier to understand how GP 2 interprets the format. It also means that it is easier to manually read a trace file if necessary, such as if the IDE crashes while trying to read the file.

A disadvantage of this format is that, because every tag has to be closed for the XML to be valid, there may be situations where the trace file saved by the running program could be invalid. For example, if the program contains a bug and fails to terminate, the user may have to kill the running program, which would cause the closing tags not to be written to the file.

There are two methods to get around this issue for non-terminating programs; the first is to add some way for the user to mark a point in the GP 2 program text at which the tracefile is finished and closed. During compilation, if the compiler sees this mark, which could be a keyword or a special kind of comment, it would insert additional code to end the tracefile at that point before continuing with execution. The user could then press `^C` at the command line to terminate the program, and continue using the tracefile. The disadvantage to this method is that it would involve modifying the specification of GP 2 syntax to include the new mark. This would result in a new compiler which is backwards-compatible, meaning that the new compiler can still compile old GP 2 code, but not forwards-compatible, because old compilers would not be able to compile programs written using the new syntax.

The second method is to add code to the compiled GP 2 program which handles the termination signal sent when the user presses `^C`. This code would print any necessary closing tags to the tracefile before exiting. This would be simpler to implement, since the compiler could statically generate the signal handling code, rather than parsing additional information from the GP 2 program text. However, it would mean that the user must attempt to guess when to press `^C`; they must wait long enough for the program to reach the infinite loop before terminating. There is no way to tell at runtime where the program has reached.

5 Implementation

5.1 Modifications to the Compiler

5.1.1 New Command-Line Argument

Since tracing is optional, the compiler needs to be told when to enable it. The simplest way to do this was to add a new command-line flag which enables tracing when it is set.

The alternative would have been to communicate this in the program text with a keyword or command to enable tracing. This would have been more complicated, because the parser in the compiler would need updating, not just the command-line interface. Additionally, there are no other keywords of this type in the GP 2 syntax, but there *are* command-line arguments for the compiler; it is best to follow the precedent when adding new features, to provide a consistent experience to users and to keep the system maintainable.

The new command-line flag, `-t`, sets a new Boolean global variable, `program_tracing`, to `true`. If the flag is not specified, the variable defaults to `false`. This global variable is then used when generating the C code to determine whether or not to add tracing code.

Care had to be taken when naming this variable, since there are some macros designed to be used for debugging the compiler itself. `GRAPH_TRACING`, `RULE_TRACING`, and `BACKTRACK_TRACING` can be set by a developer working on the compiler in order to turn on specific types of tracing. The tracing enabled by these macros is no use for program tracing, since the output is designed to be human readable, and does not contain all the necessary information to step through the execution history of a program.

5.1.2 Tracing Module

Part of GP 2's implementation is a static library named `libGP2` which contains modules for performing various tasks such as searching for nodes and edges, and modifying graph objects. It is used to avoid the compiler having to generate the same code for every compiled program, since the operations performed by `libGP2` are generic and aren't program-specific.

To implement the tracing functionality, a new module was added to `libGP2` called `tracing`, which contains functions for creating and updating a tracefile. This was created in `libGP2` because the tracing itself is not dependent on the compiled program; the act of printing to the tracefile is always the same, and all that differs are the actual details being printed.

The tracing module contains a set of functions which print data to a tracefile when called by the compiled C program. Two of these are for creating and closing the tracefile itself; these are intended to be called at the very beginning and end of the program. Two are for beginning and ending *contexts*, discussed in detail later in this section. The rest are called whenever the program modifies the graph in some way, and record details about these changes. For instance, the `traceCreatedNode()` function takes a pointer to a newly created graph node, and adds a line to the tracefile containing all the details necessary to recreate that node when stepping through the tracefile in the IDE.

Contexts

GP 2 programs have structure, including control flow and procedures. These structures can also be nested; for example the condition of a If-Then-Else statement can contain a Procedure, which itself could contain a loop, and so on. To be able to step through a user's program, the IDE needs to keep track of the structure of the program, so it can follow the execution and show the user what part of the program is being executed.

The tracing module uses the nested structure of XML documents to record the structure of a GP 2 program, by nesting GP 2 constructs into *contexts* in the tracefile. A context is simply an XML tag describing the GP 2 construct being used, such as a `<loop>` context containing the steps carried out inside an As-Long-As-Possible structure.

When the program enters a structure at runtime, it calls the `traceBeginContext()` function in the tracing module, passing the type of context as the argument. For instance, when entering a loop, the program will call `traceBeginContext("loop")`, which causes a `<loop>` tag to be printed to the tracefile.

Each context also must be closed when the program exits that structure; continuing the example from before, the program will call `traceEndContext()` when it finishes the loop. This prints the closing `</loop>` tag in the tracefile.

No argument is required when closing a context, because the tracing module keeps track of the current nesting of contexts, and always closes the most recently opened one. To track the nesting of contexts, the tracing module uses a stack. When a new context is opened, an element is pushed onto the stack which contains the context's name. When the program calls `traceEndContext()`, the topmost item is popped from the stack, and the name is read, so that the correct name is used in the closing tag.

This stack structure works because XML tags must always be closed in the reverse order to how they were opened. For instance, `<loop><procedure></loop></procedure>` would be invalid, because the tags are not closed in reverse order. It also makes sense to do this because when GP 2 structures are nested, the outer one cannot end before the inner one; a loop cannot end before a procedure inside the loop finishes.

The tracing module must keep track of contexts because when a failure occurs in the GP 2 program, no further code is executed before performing garbage collection and terminating; this means that there will possibly be contexts left open by the program. The implementation of `finishTraceFile()` uses the context stack to ensure all open contexts are closed. It pops each item from the stack in turn and closes that context, until the stack is empty and the tracefile can be closed. If contexts were left open, the XML file would be invalid and parsing would fail.

Contexts can also have a *label*, implemented using an XML attribute; most commonly this label is a name, such as the name of a procedure or rule. This is achieved by calling `traceBeginNamedContext()` and passing both the context and the name label. For example, `traceBeginNamedContext("procedure", "Colour")` would print `<procedure name="Colour">` to the tracefile.

Branch structures use a special type of label called `backtracking`, which contains either `true` or `false` depending on whether backtracking is required for that structure. As discussed in Subsection 4.1.2, in some cases the compiler does not generate backtracking code because the rules applied in the conditional part of the structure do not make changes to the graph. In this case, the `backtracking` label is set to `false` so that the IDE knows the changes do not need to be reversed once tracing reaches the end of the condition. This is achieved using the `traceBeginLabelledContext()` function.

5.1.3 Changes to Compiled Code

When tracing is enabled in the compiler using the previously dicussed command-line argument, additional code is generated in the compiled C program to call the functions in the tracing module at the correct times.

At the beginning of the generated `main()` function, the compiler generates a call to `beginTraceFile()`, which creates a new file in the same directory as the output graph, and starts the file by printing the XML doctype specifier, and an opening `<trace>` tag containing the name of the program being run, and the filename of the input graph.

There is a matching call to `finishTraceFile()` in the generated `garbageCollect()` function. This function is called whenever the program is about to exit, meaning either the end of `main()` is reached, a rule fails to match, or the GP 2 `fail` statement is used. This call could not simply be put at the end of `main()`, since the compiled program exits early if a failure occurs. `finishTraceFile()` must always be called, no matter how the program exits, because the tracefile must contain a closing `</trace>` tag for the XML to be valid. If it is invalid, the IDE will fail to parse the file, and it will not be able to step through the trace.

In the compiled program, the `main()` function contains the high level structure of the program; rule matching and application are handled by functions called from `main()`. In the code to generate `main()`, additional code was added at each point where a control structure starts or ends. For example, Listing 5.1 shows pseudocode for generating C code to run a GP 2 Procedure.

Listing 5.1: Pseudocode for generating a Procedure

```
if (GP 2 structure is of type Procedure) {
  if (program_tracing) {
    generateCode(traceBeginNamedContext("procedure",
        procedure.name));
  }
  generateProcedureCode(procedure);
  if (program_tracing) {
    generateCode(traceEndContext());
  }
}
```

The only changes to the original code are the new conditional statements which check whether tracing is enabled, and if it is, make the calls to begin and end the `<procedure>` context. The changes to all other GP 2 structures are similar, simply adding two extra function calls if tracing is enabled. Because such a small amount of code was added to the compiler in each location, it is still easy to understand the code, and to see which lines are related to tracing.

The other change is when generating the functions for matching and applying rules. In the case of matching, the function `traceRuleMatch()` is called, passing a pointer to a `Morphism` object, which contains a list of the nodes, edges, and variables used in a rule match. The tracing module then iterates over the objects in the `Morphism` and prints them into the tracefile inside a `<match>` context. The function also takes a Boolean argument specifying whether the match was successful or not; this is required since an empty `Morphism` can either mean the match failed, or there are simply no nodes or edges in the match (the LHS of the rule is empty).

During a rule application, a tracing function call is generated each time a change is made to the graph. For instance, this occurs when an edge or node is deleted from the graph, or when a node is relabelled. There is a function for each type of graph operation in the tracing module, and the generated code calls the correct one in each case.

In each case, it is simply one or two lines of code inside a conditional statement checking whether tracing is enabled. The addition of tracing to the compiler did not have a large impact on the amount of code required, except for the addition of the tracing module itself in `libGP2`. However, since all the tracing code is contained in that module, it is easy to remove if necessary, and the existing compiler code can function independently of having the module present, as long as the `program_tracing` variable is set to `false` when the tracing module is missing.

5.2 Changes to the IDE

5.2.1 UI Changes

There are two changes in the IDE's UI. First, a checkbox has been added for each run configuration in the Run tab, shown in Figure 5.1. Enabling this checkbox before running a run configuration sets the new compiler command-line argument to enable tracing for that run. If the checkbox is disabled, the program runs without tracing, resulting in identical behaviour to before the IDE was modified. This means the user is free to use the IDE without being forced to run tracing.

The other modification was to add a new tab to the left side of the window which, when clicked, opens a screen where the user can view and control the trace. This screen is shown in Figure 5.2.

At the top of this screen is the graph at the current point in the trace. This graph view is updated whenever the user steps forward or backward in the trace, to reflect the graph as it would have appeared if the program ended at that point.

Below the graph view is a status bar, a small line of text which is updated to give the user information about the state of the program. For example, when a rule fails to match, a message is displayed in the status bar. The status bar is necessary because some situations, such as failed rule matches, cannot be displayed on the graph or in the program text, since there is no well-defined or clear way to do so.

Next is a row of buttons which are used to control the trace. It is possible to step forward and backward by one step, or to jump directly to the beginning or the end of the trace. There is also a button which highlights a rule match, if there is a valid match. The match is shown in the graph view by highlighting the nodes and edges which comprise the match, as shown in Figure 5.3. The button then changes to allow the user to apply the rule on the match being shown.

Finally, at the bottom of the screen, there is a copy of the program text. In this section, the next step of the program is highlighted with a green background. This shows the user what will be executed if they press the "Step Forward" button. Some IDEs, such as Visual Studio, display a marker on the left edge of the program text to show the next line of code which will be executed. This would not work for GP 2, because there may be multiple rule calls on a single line of a GP 2 program, and it is important to distinguish which one being called. This is why the individual program tokens must be highlighted, rather than the entire line.

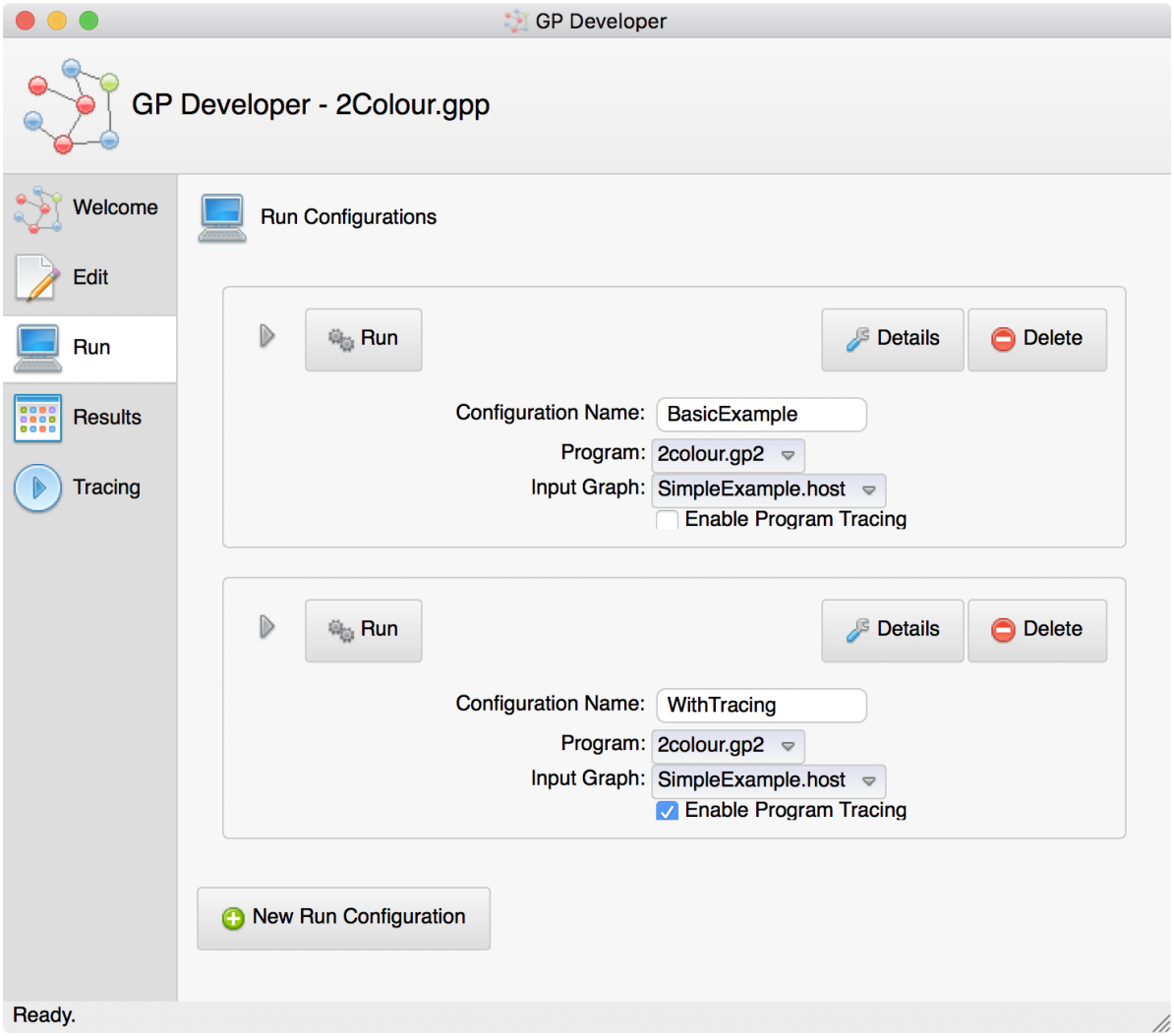

Figure 5.1: Checkbox to enable tracing for a run configuration

Appendix A contains a sequence of screenshots displaying the tracing UI as it changes during an entire trace of the 2-colouring program.

5.2.2 Tracing Backend

To support reading tracefiles and displaying the trace in the UI, five new components were added to the IDE's code. These components were designed using the Model-View-Controller (MVC) pattern [20]. The *model* reads the tracefile and tracks the state of the trace, and is formed of three of the five new components. The other two components are the *view*, which controls what is shown on-screen, and the *controller*, which handles communication between the view and the model.

Using MVC makes the system easier to maintain and modify; for example, if a bug is discovered in the model related to parsing the tracefile, the view component does not need to be modified, since the view simply updates the user interface based on the data it receives from the controller.

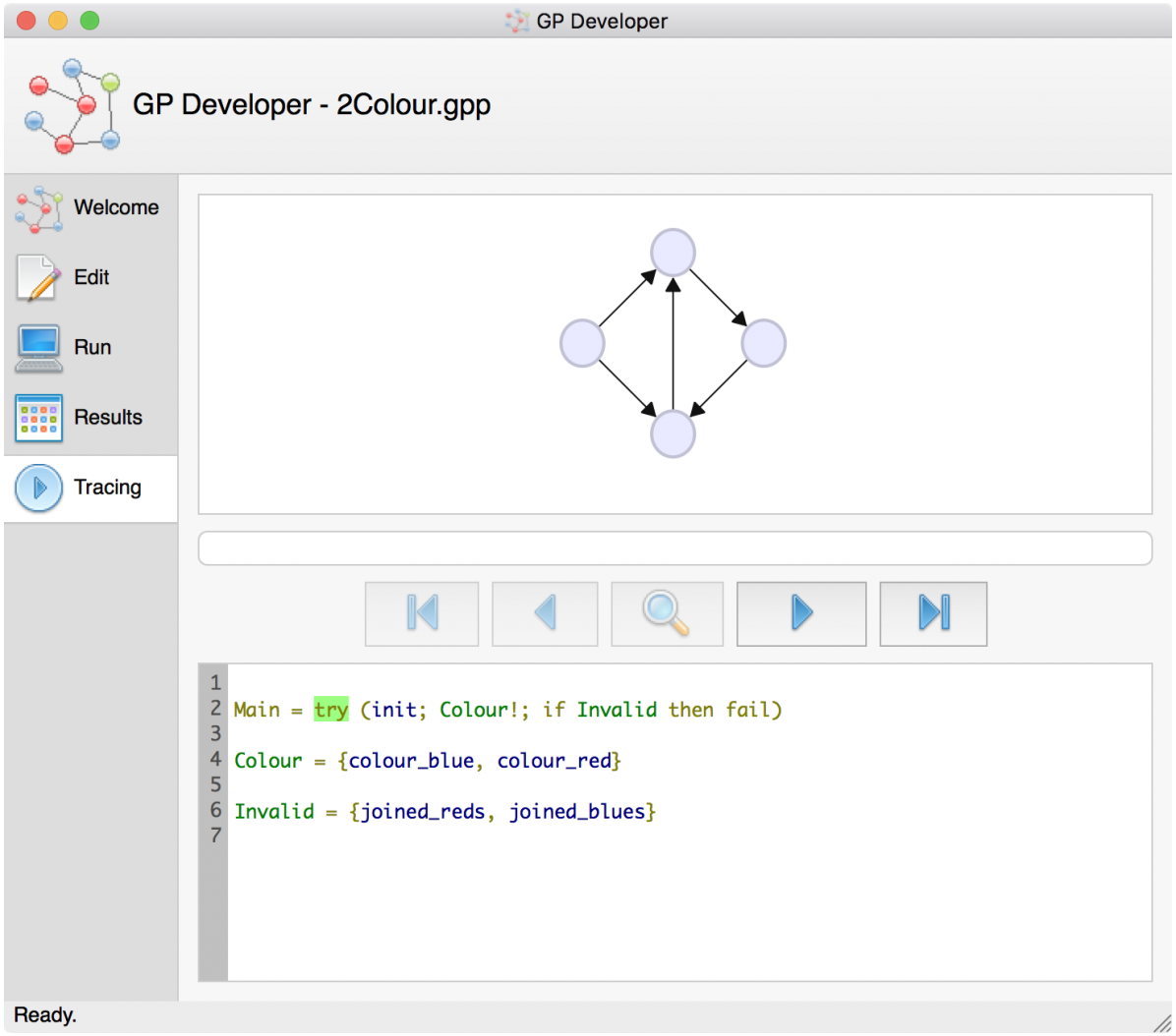

Figure 5.2: Tracing tab in the IDE

Model

The model is made up of three C++ classes, each designed to handle a specific task. This modularity means that functionality is encapsulated within an object designed to perform that specific function, increasing the readability and maintainability of the system.

- The `TraceParser` class is responsible for reading the tracefile and determining the steps the program took while it was running. It is implemented using the Qt class `QXMLStreamReader`, which parses an XML document from a data stream; it does not require an entire XML file to begin the parsing process. This is useful because it means the IDE can handle tracing even from a program which crashed or was forced to terminate early, even though the end of the tracefile will be missing in these cases.

  Each time the `TraceParser` is called, it parses one XML tag, such as a `<procedure>` context, or a graph change such as a `<createdNode>` element. This tag is then converted into a custom C++ `TraceStep` structure which contains all the information about that step in the trace, including the type of step, any changes made to the graph

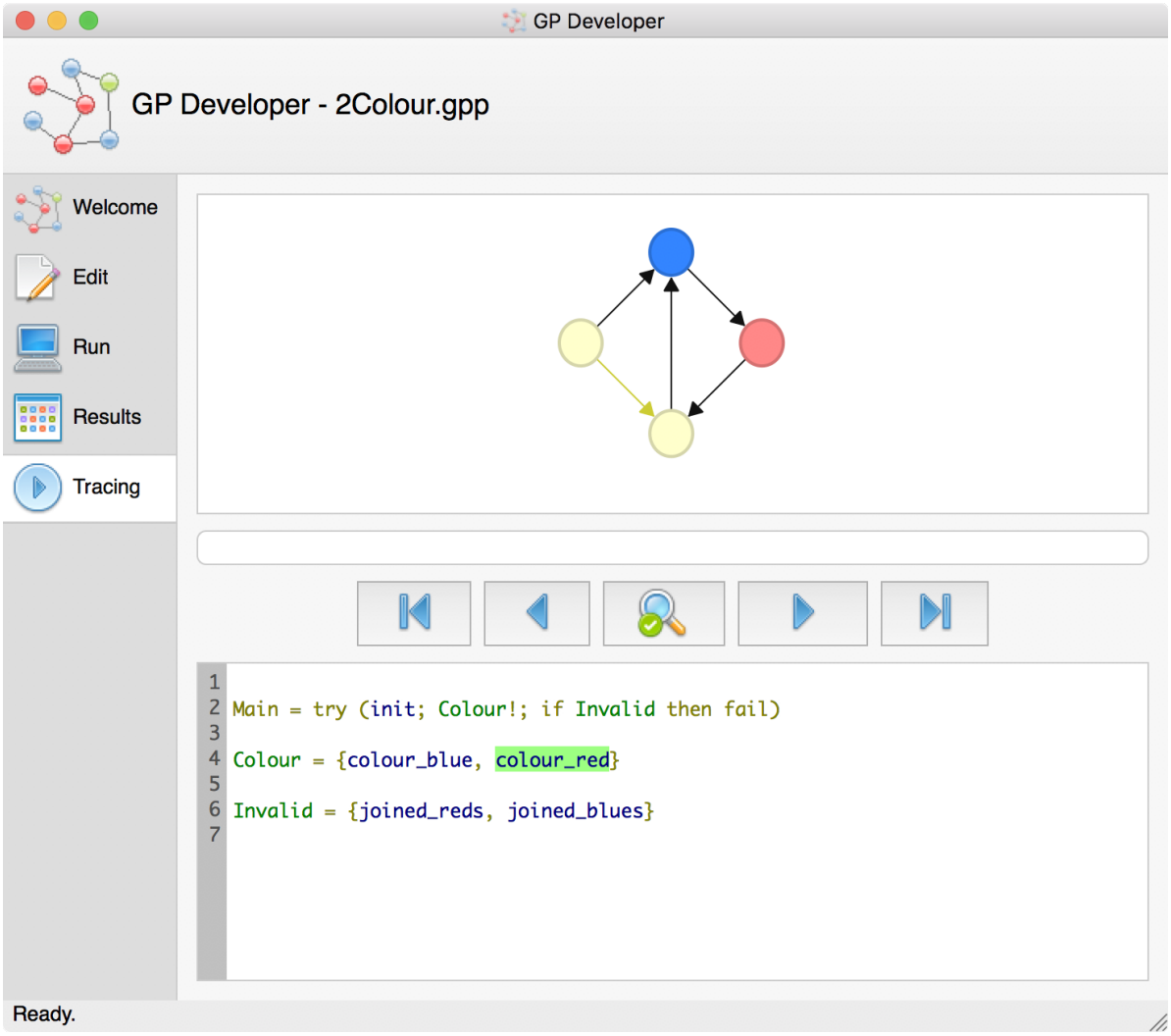

Figure 5.3: Match highlighted in the tracing graph view

during that step, and the name of the context. These structures are then stored in a list, in order, so that iterating over the list steps through the trace.

- The `TraceHighlighter` class is used to determine which part of the program text should be highlighted in the UI. Its `update()` method takes a `TraceStep` structure as an argument, which is used to find the matching portion of the program text.

  For instance, if it is passed a `TraceStep` object representing a rule call, it starts from the current highlighted position and searches through the program looking for a call to a rule with the same name. Since GP 2 programs are executed in textual order, the first matching rule call it finds must be the correct one.

  When a `TraceStep` is passed in which represents the beginning or end of a context (as defined in Section 5.1.2), it searches for GP 2 keywords, such as `if`, `try`, or `skip`, or symbols, such as the loop operator `!` or curly braces `{ }` for a ruleset.

- Finally, the `TraceRunner` class stores the state of the trace, and calls the `TraceParser`

and `TraceHighlighter` classes when required. It is this class which handles method calls from the controller, either to step through the trace in a specified direction or to find and highlight a match for the current rule.

When the `step()` method is called, it attempts to load the next `TraceStep` structure from the list. If there are none left in the list, the `TraceParser` class is called in order to parse the next step of the trace. The parsing is done incrementally in this way to increase performance and reduce memory usage; parsing the entire tracefile may take a significant amount of time if the program had a large number of steps, and each of these steps would need to be stored in the list. By parsing incrementally, trace steps are only parsed and stored when the user reaches them in the trace, so if they exit the trace before the end, the parser does not waste time parsing the rest of the XML.

After loading the next `TraceStep`, any graph changes from that step are applied to the graph. If the user is stepping backwards through the trace, the graph changes from the step are reversed; for example, if a node was added at that point, stepping backwards would remove the node from the graph. The `TraceRunner` class exposes a `getGraph()` method to return the current graph state for displaying in the UI; the controller calls this method when the user clicks a button to step through.

Finally, the `TraceStep` *after* the current one is then passed to the `TraceHighlighter`, which returns the position in the program which should be highlighted. As previously discussed, the next step is highlighted, not the step which was just executed, so that the user is aware of what will occur when they click the step button.

View

The view is a Qt *form*, a special file format which tells the Qt framework which UI elements to display and how to display them. The view component does not contain any code; it simply describes how to build the UI as outlined in Subsection 5.2.1.

Controller

The controller is a small C++ class which acts as a communication layer between the view and the model. When the user clicks a button in the view, the controller forwards a method call onto the model, performing the action intended by the button which was pressed.

At the end of the method call linked to each button, the controller fetches the most recent graph and program highlight state from the `TraceRunner`, and uses it to update the view.

6 Evaluation

6.1 Requirements Tracing

To evaluate the new functionality added to GP 2, the implementation has been compared to the list of requirements outlined in Section 3.4. Table 6.1 lists each of the requirements, along with whether that requirement has been met by the system.

All of the must-have system requirements have been implemented, meaning that the system successfully implements a debugging tool for GP 2 programs. The two unimplemented requirements were defined as "desirable" and were not essential for the tool to be useable.

6.2 Maintainability

Some key goals were set out in Section 3.2 related to the maintainability of the system. Throughout the project, any modifications being made to the system were designed with ease of understanding in mind. The newly added code follows the conventions set out by the existing GP 2 code, and all new code has comments. Semantic names were chosen for variables, functions, and methods to ensure future developers who are unfamiliar with the changes can understand what the code does.

Careful thought was applied to ensure that tracing was optional when compiling and running a GP 2 program, and that the new code did not create strong dependencies between components. This has been achieved by separating the tracing components into separate modules, both in the compiler and the IDE, meaning they could easily be removed or changed without affecing the behaviour of the rest of the system.

6.3 Performance

It was important that the changes made to GP 2 did not negatively impact the speed of compiling and running a program, since it would make debugging a program cumbersome to have to wait longer for the program to execute.

To check the performance of the system, the Sierpinski Triangle generation program mentioned in Section 4.3 was compiled and run with tracing disabled and then with tracing enabled, to compare the time taken in both cases. The program was set to generate the 11th-generation Sierpinski Triangle. This program is expected to take between 10 and 15 seconds, due to the algorithm requiring exponential time $O(2^n)$ in the generation number.

In each case, the program was compiled and run ten times, and the average time was taken. These tests were run on a PC with a 4.0 GHz processor and 24 GB of RAM using the UNIX `time` facility to measure the time taken.

The results of these tests are shown in Table 6.2. There is very little difference between the compilation times, as expected, since compiling a program with tracing simply adds some additional function calls to the compiled C program.

At runtime, the program with tracing enabled runs for approximately 17% longer than the program with tracing disabled. This is due to the program writing trace data to the

ID	State	Summary
1	Implemented	A GP 2 program can be traced from start to end, producing the same result as the original output.
1.1	Implemented	The "Step Forward" button applies one rule at a time.
1.2	Implemented	The trace can be reversed to step backwards through the program.
1.3	Not Implemented	It is only possible to step rule-by-rule, or to jump to the beginning or end of the trace. There is no way to execute an entire loop, for instance.
2	Implemented	The GP 2 IDE contains a Tracing tab which is used to visually trace a program's execution.
2.1	Implemented	Rule calls are highlighted in the program text when they are executed in the trace.
2.2	Implemented	The "Find Match" button highlights the match for the current rule. This is a manual operation; the match is not highlighted for every step unless requested by the user.
2.3	Implemented	The program text is highlighted in the Tracing tab to show the current position.
2.4	Not Implemented	The system has no way to highlight which changes were made by the previous rule application.
2.5	Implemented	Whenever the graph changes, the latest graph is shown in the IDE.
3	Implemented	The user can control tracing using buttons in the tracing tab of the IDE.
3.1	Implemented	The "Step Forward" button advances the trace by one step.
3.2	Implemented	The "Step Backward" button reverses the trace by one step.
3.3	Implemented	The "Find Match" button highlights the match for the next rule.
3.4	Implemented	When the "Find Match" button is used, it becomes an "Apply Match" button, which completes the application of the rule.
3.5	Implemented	Tracing can be disabled using the checkbox in the Run tab.
4	Implemented	GP 2 programs behave identically when tracing is enabled and disabled, since all code changes were additive and did not modify existing behaviour.
4.1	Implemented	Enabling tracing simply outputs a tracefile during execution, and does not change the behaviour of the program. The results are identical whether tracing is enabled or disabled.
4.2	Implemented	When old projects are loaded, tracing is disabled by default for each run configuration. Nothing else needs to be modified to run old programs.

Table 6.1: State of each of the previously defined system requirements

filesystem at each step. A performance decrease of 17% is fairly small, and would probably not be noticeable for small programs with short runtimes. For larger programs which may take a minute or longer to run, the performance impact may be concerning, since it would be much more noticeable.

Tracing	Average Compilation Time (ms)	Average Run Time (s)
Disabled	9.6	12.970
Enabled	10.0	15.180

Table 6.2: Average performance of the 11th-generation Sierpinski Triangle program

It may be desirable for future work to be done on the tracing component in order to reduce the performance impact. However, as an initial implementation, a 17% impact is acceptable, and the system is still reasonable to use, especially for small GP 2 programs.

7 Conclusions

In this project, a need was identified for a way to closely follow the execution of a GP 2 program, in order to find bugs or improve understanding. To fulfil this requirement, the ability to trace a GP 2 program was added to the compiler, and a way to view and control traces was added to the IDE.

When tracing is enabled in the compiler, additional function calls are added to the compiled program, each of which prints some data about the running program to an XML tracefile. Enough of these function calls are made throughout the execution of the program so that the tracefile contains enough information to fully recreate the actions taken by the program, after the program has finished running.

In the IDE, a new module was added which parses the tracefile created by the program, and uses this to display the intermediate states of the program to the user. The user is able to step through the trace, both forwards and backwards, and see how the graph changes as the program progresses. The tracing UI in the IDE also highlights the current position in the program text, and has the ability to highlight the nodes and edges in the graph which make up the match for a given rule call.

The newly implemented system meets all the must-have requirements to be able to fully trace a program's execution. It also meets some additional non-essential requirements, such as being able to step backwards, to allow the user to see part of the program again without having to restart from the beginning.

Overall, this project has been successful, and has achieved the goals which were set out. The tracing system is entirely functional and brings substantial improvements to the way GP 2 programs can be reasoned about and debugged.

7.1 Future Work

Whilst the system is fully functional, there are still a number of areas that have been identified where further work could improve on the system as it is at the end of this project. These improvements could not be implemented in this project due to time constraints.

7.1.1 User Feedback

As part of the evaulation of the system, it was originally intended that a small subset of potential users would be shown the implemented tracing system and asked for some feedback or possible improvements. Due to the time constraints, it was not possible to arrange for users to test the system before the end of the project.

It would be beneficial to get feedback from users, since it would give insight into how the system is used in the real world, and would potentially find issues with the user experience, such as confusing behaviour, which are not apparent from developer testing.

7.1.2 Performance

As described in Section 6.3, enabling tracing has around a 17% impact on the time it takes to run the Sierpinski Triangle program. It should be expected that the performance of other programs is also similarly affected. This is due to the program writing to the filesytem every time trace data is collected.

To improve the performance when tracing is enabled, writing to the tracefile could be performed in batches. Trace information would be stored in memory until either a specified amount of data is collected, or until a specified point in the program is reached, such as the end of a procedure or the end of a loop iteration. At that point, all the trace information in memory would be written to the tracefile at once. This should decrease the amount of time spent waiting for the filesystem and therefore decrease the overall runtime of the program.

7.1.3 Skipping Over Program Structures

One of the nice-to-have requirements was to allow the user to execute an entire control sequence as one step, such as executing an entire loop or procedure without stepping through the constituent parts.

Implementing this feature would be fairly straightforward, since the tracefile uses nested *contexts* (described in Section 5.1.2). To skip over a procedure, loop, or any other structure would simply require reading through the parsed `TraceStep` objects in the `TraceRunner` until the end of the current context is reached.

For example, if the trace is currently in a `<loop>` context, skipping over the loop would involve iterating over the `TraceStep` structures until the closing `</loop>` step is found. Care would also have to be taken if there is a nested loop, making sure to only stop iterating when the *current* loop ends, not any inner loops, since there would be additional `</loop>` steps in that case.

7.1.4 Highlighting Graph Changes

Another useful feature would be the ability to highlight in the tracing view any changes which were made to the graph by the previous step. In a large graph, this would make it easier to see exactly what was changed by a rule.

This would be another simple feature to implement, since each `TraceStep` structure contains a list of changes made to the graph at that step. All that would be required is iterating over this list after executing the step, and highlighting all the nodes and edges referenced in that list.

It would not be possible to highlight deleted nodes and edges, since they would no longer appear in the graph. In that case, it may be preferable to list the deleted objects in the status bar below the graph.

7.1.5 Breakpoints

The final possible improvement that was identified was the addition of breakpoints. This would allow the user to mark a position in the program and have the trace run until that point. This would improve the user experience of debugging, because it means the user would not have to repeatedly click the “Step Forward” button to reach the point in the program they are interested in.

This would require more substantial work to implement than the previous features. There are two possible approaches to implementing breakpoints within the current architecture of the system.

The first would be to add a keyword to the GP 2 syntax which represents a breakpoint. When compiling a program containing this keyword, a function call would be generated to print a breakpoint into the tracefile. This would then appear as a special form of `TraceStep` in the IDE, and the `TraceRunner` could iterate over steps until it reaches that special step. This method would be a larger change, since it would require modifying the GP 2 parser, lexer and compiler, as well as the IDE.

The other option would be to allow the user to select a token in the program text in the tracing tab, and iterate through the trace until the `TraceHighlighter` highlights the selected token. This is a more lightweight change than adding a keyword to the language, but would require more work in the tracing UI, to add the ability to select a program token.

Bibliography

[1] 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*, vol. 14, no. 1, pp. 15–40, 2012.

[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] 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.

[4] D. Plump, "The design of GP 2," 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.

[17] 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)*, ser. Electronic Proceedings in Theoretical Computer Science, vol. 181, 2015, pp. 48–64.

[18] C. Bak and D. Plump, "Compiling graph programs to C," in *Proc. International Conference on Graph Transformation (ICGT 2016)*, vol. 9761. Springer, 2016, pp. 102–117.

[19] G. Taentzer, E. Biermann *et al.*, *Generation of Sierpinski Triangles: A Case Study for Graph Transformation Tools*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 514–539.

[20] G. E. Krasner, S. T. Pope *et al.*, "A description of the Model-View-Controller user interface paradigm in the Smalltalk-80 system," *Journal of object oriented programming*, vol. 1, no. 3, pp. 26–49, 1988.

Appendices

A Appendix A: Full Program Tracing Sequence

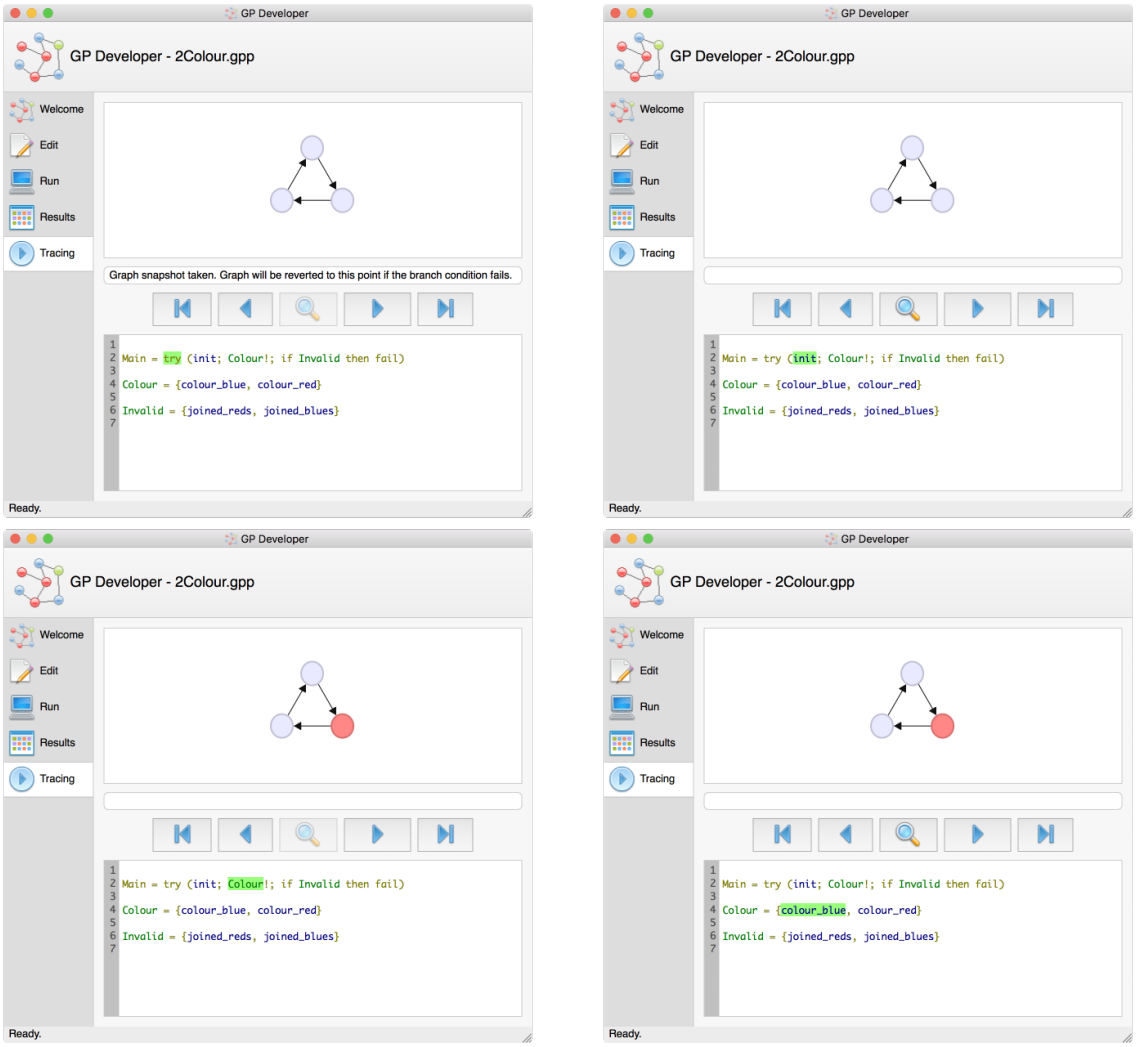

Figure A.1: Full program tracing sequence

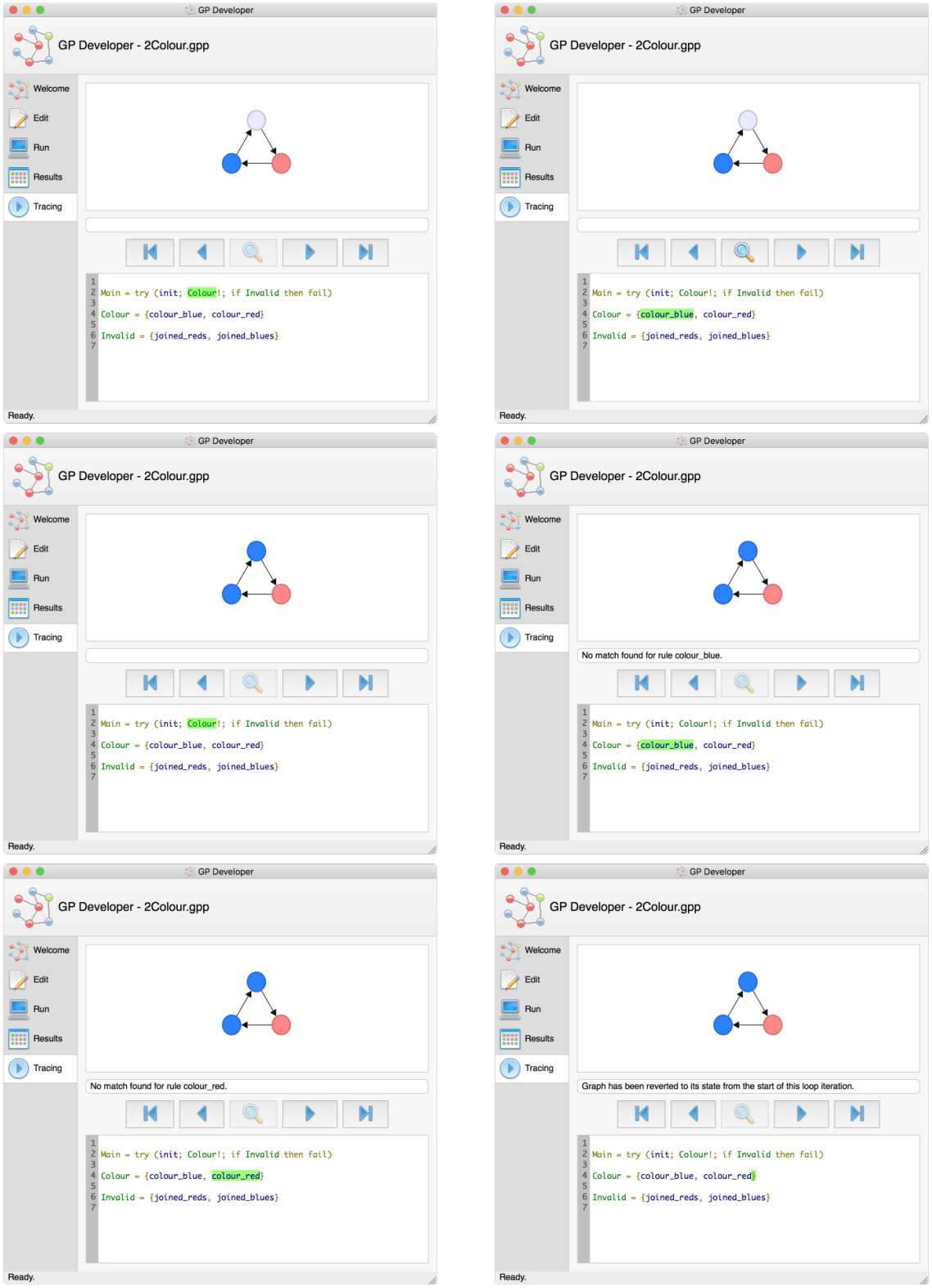

Figure A.1: Full program tracing sequence (cont.)

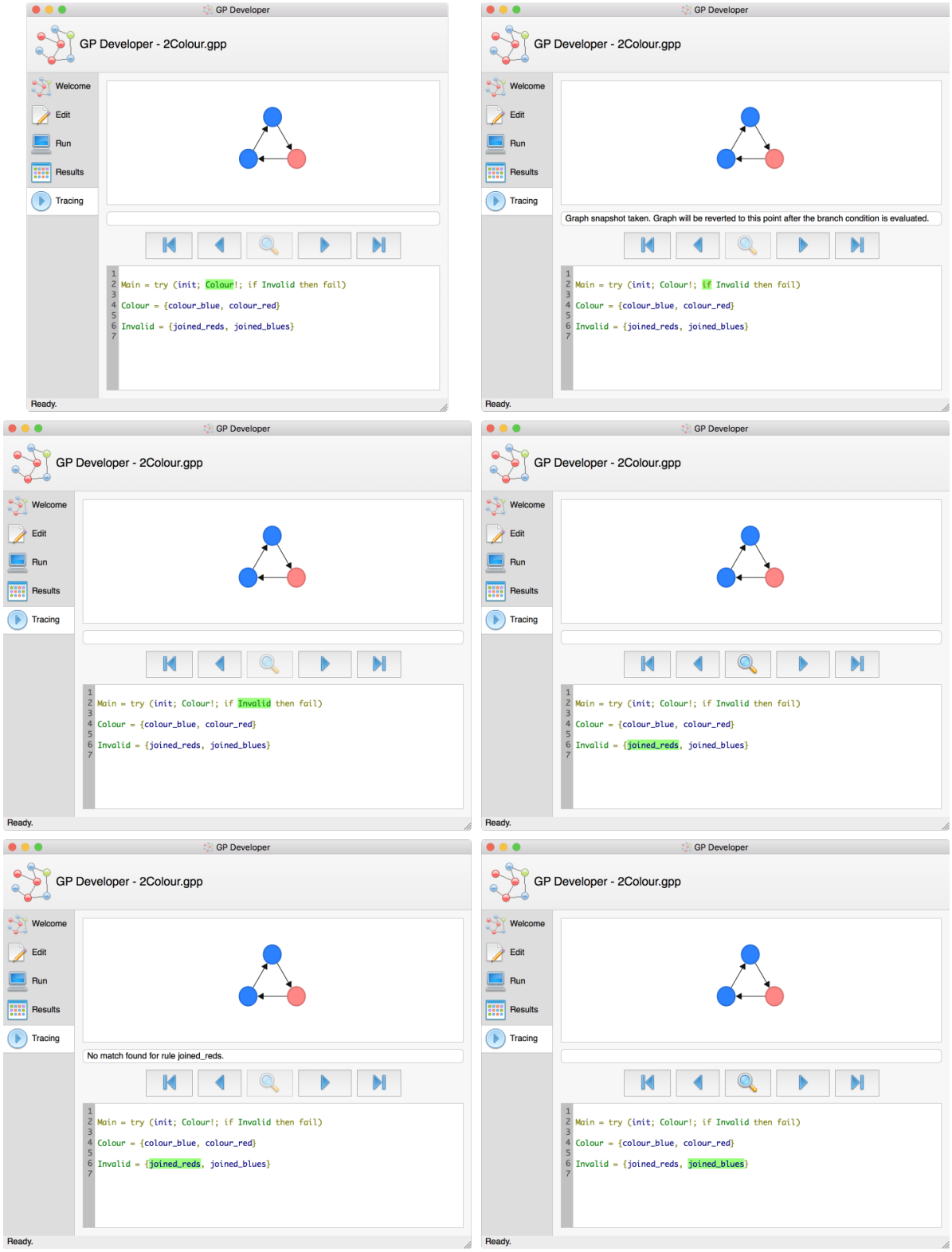

Figure A.1: Full program tracing sequence (cont.)

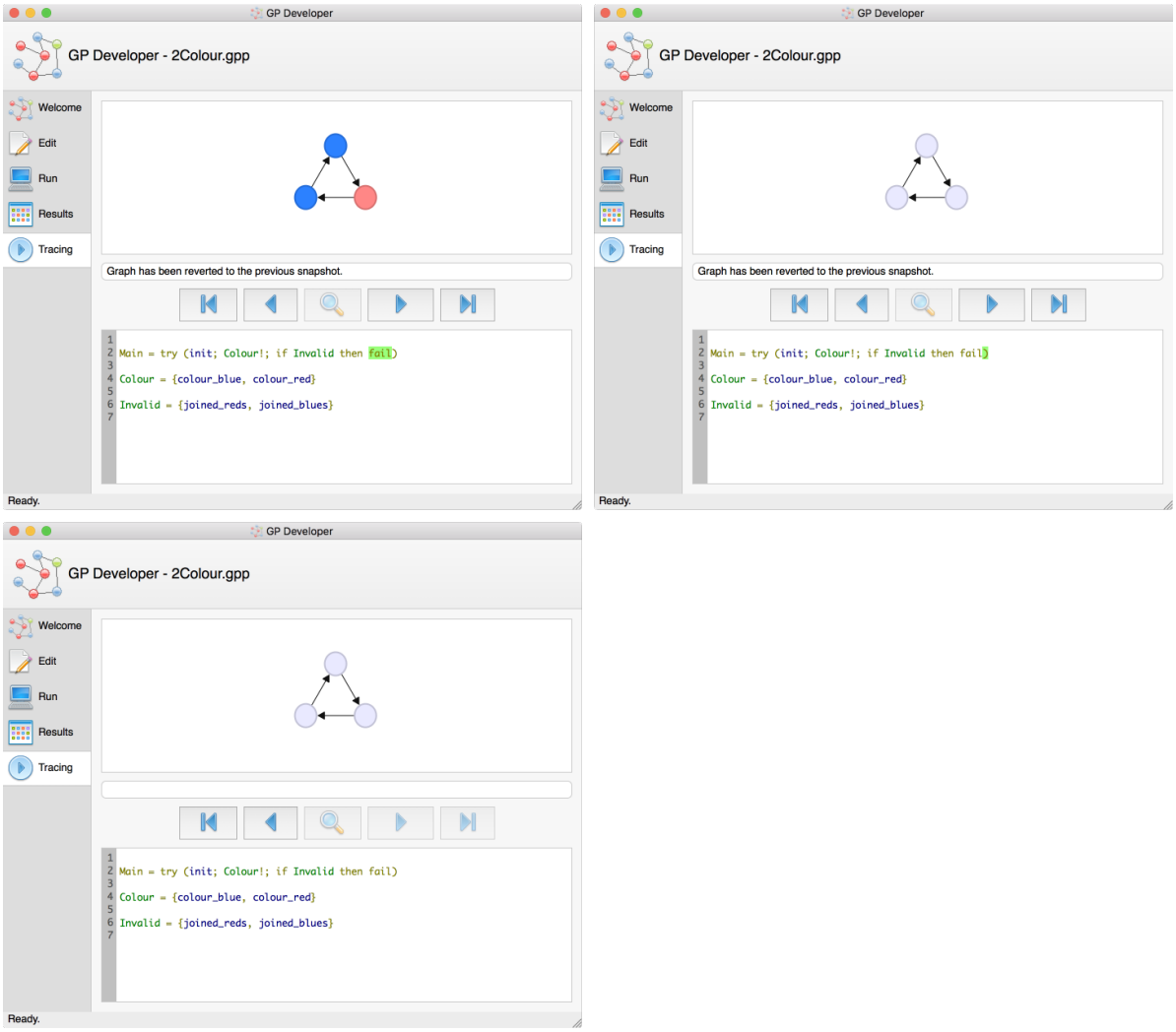

Figure A.1: Full program tracing sequence (cont.)