A Reference Interpreter for the Graph Programming Language GP 2

Christopher Bak,* Glyn Faulkner,* Detlef Plump and Colin Runciman

Department of Computer Science, The University of York, UK

GP 2 is an experimental programming language for computing by graph transformation. An initial interpreter for GP 2, written in the functional language Haskell, provides a concise and simply structured reference implementation. Despite its simplicity, the performance of the interpreter is sufficient for the comparative investigation of a range of test programs. It also provides a platform for the development of more sophisticated implementations.

1   Introduction

GP 2 is an experimental programming language in which the major part of the computational state is a labelled directed graph, and the basic units by which computational progress is made are subgraph-replacement rules. Choices of rules and subgraphs are non-deterministic, and some of the control structures above the level of rules involve back-tracking.

The implementation of such a programming language poses some interesting challenges and opportunities. Our ultimate goal is to produce a compiler from GP 2 to high-performance executable code. This paper reports a first stage towards that goal, the development of a *reference interpreter* for GP 2. By this we mean an interpreter written with the main aim of being clear, concise and correct. Where there are design choices, simplicity of definition takes priority over other considerations such as performance and the richness of functionality. The interpreter contains only around 1,000 lines of Haskell source code. Even so, we shall show that it is usable in practice.

Section 2 outlines and illustrates the graph programming language GP 2. Section 3 presents a small set of test programs written in GP 2. Section 4 considers the expected uses of a reference interpreter, and consequent requirements. Section 5 describes our reference interpreter for GP 2. Section 6 sets out the measured results of using the reference interpreter to evaluate test programs. Section 7 briefly discusses related work and indicates some of our own expected lines of future work. Section 8 draws overall conclusions from our work on the reference interpreter for GP 2.

2   Graph Programs

This paper focusses on GP 2, a successor to the graph programming language GP [14, 15]. GP is a domain-specific language which aims to support formal reasoning on graph programs (see [16] for a Hoare-logic approach to verifying GP programs). We give a brief introduction to GP 2, mainly by example. The definition of the language, including a formal operational semantics, can be found in [15].

A graph program consists of declarations of conditional graph transformation rules and macros, and exactly one main command sequence. Graphs are directed and may contain loops and parallel edges. The rules operate on a *host graph* (or input graph) whose nodes and edges are labelled with a list of integers

---

and character strings. Besides the list, a label may contain a *mark* which is one of the values `red`, `green`, `blue`, `grey` and `dashed` (where `grey` and `dashed` are reserved for nodes and edges, respectively). For example, the node label on the right-hand side of the rule `init` in Figure 2 is the pair $\langle$`x:1`, `grey`$\rangle$.

Variables in rules are of type `int`, `char`, `string`, `atom` or `list`, where `atom` is the union of `int` and `string`. Atoms are considered as lists of length one, hence integers and strings are also lists. Similarly, characters are considered as strings of length one. Given lists `x` and `y`, their concatenation is written `x:y` (not to be confused with the list-cons operator in Haskell).

Example 1 (Transitive Closure). The principal programming constructs in GP 2 are conditional graph-transformation rules labelled with expressions. The program in Figure 1 applies the single rule `link` *as long as possible* to a host graph. In general, any subprogram can be iterated with the postfix operator "!". (A composite loop $(P_1; \ldots; P_n)!$ terminates if any of the components P_i *fails*, meaning that some rule in P_i could not be matched. In this case the loop finishes with the graph on which the current iteration of the body $(P_1; \ldots; P_n)$ was entered. See [15] for details.)

$$\texttt{Main} = \texttt{link!}$$

$$\texttt{link(a,b,x,y,z: list)}$$

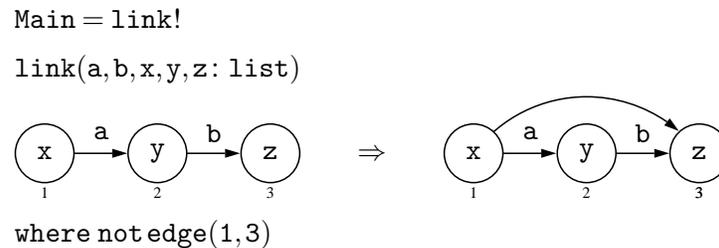$$\texttt{where not edge(1,3)}$$

Figure 1: Program for transitive closure

Applying `link` amounts to non-deterministically selecting a subgraph of the host graph that matches `link`'s left graph, and adding to it an edge from node 1 to node 3 provided there is no such edge (with any label). The application condition ensures that the program terminates and extends the host graph with a minimal number of edges. Rule matching is injective and involves instantiating variables with concrete values (see also below).

A graph is *transitive* if for each directed path from a node v to another node v', there is an edge from v to v'. Given any graph G, the program in Figure 1 produces the smallest transitive graph that results from adding unlabelled edges to G.[1] This graph is unique up to isomorphism and requires at most n^2 applications of `link`, where n is the number of nodes in G. □

Example 2 (Vertex Colouring). The program in Figure 2 assigns a *colour* to each node of the host graph, such that non-loop edges have differently coloured endpoints. Positive integers are used as colours because, in general, an unbounded number of colours is needed. The program replaces each node label l with $l{:}i$, where i is the node's colour. In addition, the rule `init` shades nodes to prevent repeated application to the same node.

Rule `inc` is applied to the host graph as long as there are edges with identically coloured endpoints. It can can be shown that this terminates after at most n^2 rule applications, where n is the number of nodes. In contrast to the previous example program, *different graphs may result* from this process. In particular, there is no guarantee that the number of colours produced is minimal. For instance, Figure 3 shows two different colourings produced for the same host graph. □

---

[1] "Unlabelled" edges are actually labelled with the empty list.

```
Main = init!; inc!
```

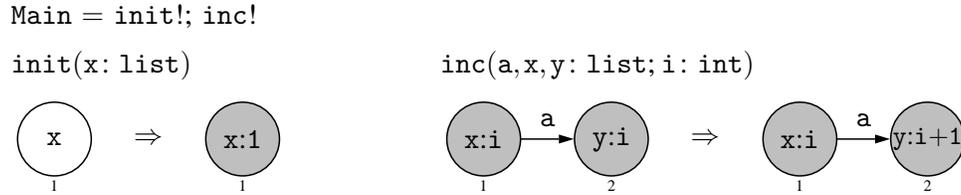init(x: list)   inc(a,x,y: list; i: int)

Figure 2: Program for vertex colouring

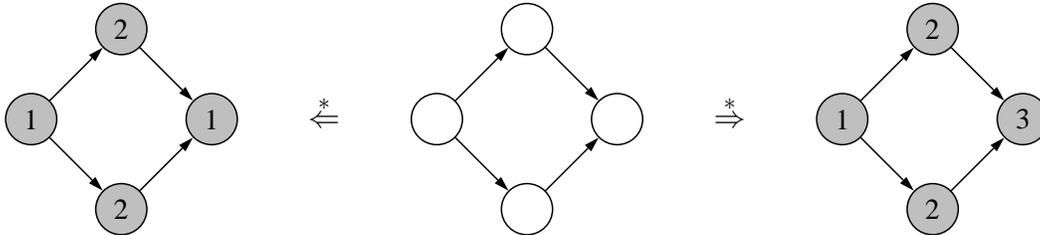Figure 3: Different results from vertex colouring

Other program constructs. A GP 2 command not used in the example programs is a rule set $\{r_1, \ldots, r_n\}$. This command non-deterministically applies any of the rules to the current host graph. The application *fails* if none of the left-hand graphs in the rules matches a subgraph. Matches must be injective and are only valid if they do not result in *dangling edges*. (More formally, GP 2 is based on the double-pushout approach with injective matching, extended with relabelling and rule schemata [15].)

Another construct not yet discussed is the branching command if C then P else Q, where C, P and Q are arbitrary command sequences. This is executed on a host graph G by first executing C on a copy of G. If C succeeds, P is executed on the original graph G; otherwise, Q is executed on G. The command try C then P else Q has a similar effect, except that P is executed on the graph resulting from C's execution.

3   Benchmark Programs

We envisage GP 2 as a general-purpose language for graph problems, hence the reference interpreter should be tested on algorithms of varying complexity. This is different from the benchmarking reported in [20] which focusses on a deterministic program with very limited complexity. In Section 6, we evaluate the performance of our interpreter on a small set of benchmark programs. These include the programs for transitive closure and vertex colouring, and three more programs which we describe in this section.

Shortest distances. The program in Figure 4 expects an input graph G containing a unique grey node s, where edge labels are assumed to be non-negative integers. A unique output graph is obtained by marking grey each node reachable from s and replacing its label l with $l:d$, where d is the shortest distance from s. (A distance is the sum of the edge labels of a directed path.)

The program first assigns distance 0 to the unique start node s. Then the loop add! traverses the nodes reachable from s, assigning distances by adding edge labels. In a second phase, the loop reduce! minimizes distances by searching for edges whose sum of source node distance and edge label is smaller
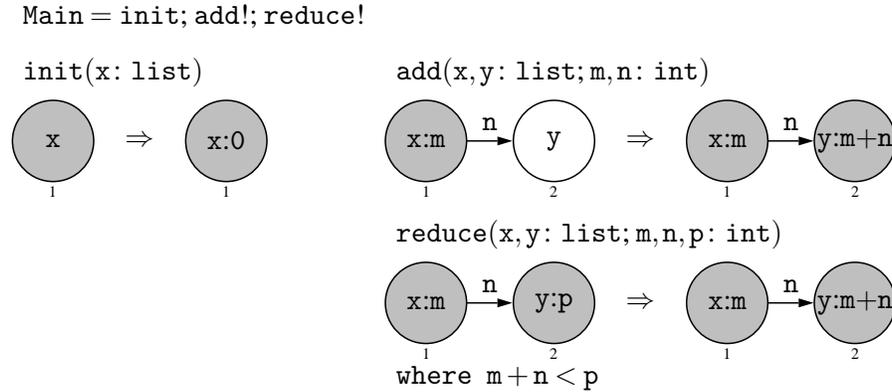
```
Main = init; add!; reduce!
```

init(x: list)

add(x, y: list; m, n: int)

reduce(x, y: list; m, n, p: int)

where $m + n < p$

Figure 4: Program for shortest distances

```
Main = if Cyclic then fail
Cyclic = delete!; {edge, loop}
```

delete(a, x, y: list)

where $indeg(1) = 0$

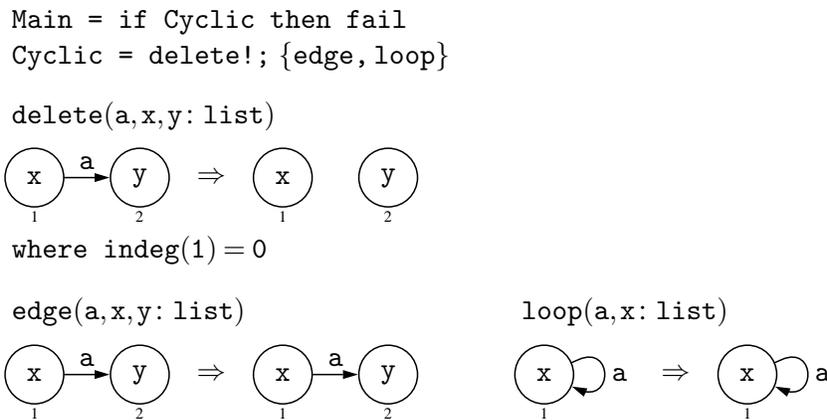edge(a, x, y: list)   loop(a, x: list)

Figure 5: Program for recognising acyclic graphs

than the target node distance, and replacing the target node distance with the sum.

The requirement that edge labels are non-negative ensures that the program terminates. It can be relaxed by allowing negative edge labels but requiring that directed cycles have a non-negative overall distance.

Recognising acyclic graphs. The program in Figure 5 checks whether its input graph is acyclic. If this is the case, the program preserves its input graph, otherwise it fails. Suppose we call the program `acyclic` to use it as a macro in the program `if acyclic then` *P* `else` *Q*. Given any input graph *G*, this program will test whether *G* is acyclic and, depending on the result, either execute *P* or *Q* on *G*.

The presence of cycles is checked by deleting as long as possible edges whose sources have no incoming edges, and testing whether any edges remain. This is correct since an application of `delete` preserves both the absence and the presence of cycles (by the condition of the rule). Moreover, a graph to which `delete` is not applicable is acyclic if and only if it is edge-less (every acyclic graph with edges must contain an edge to which `delete` is applicable).

Generating Sierpinski triangles. A *Sierpinski triangle* is a self-similar geometric structure which can be recursively defined. Figure 7 shows a Sierpinski triangle of generation three, composed of three

second-generation triangles, each of which consists of three triangles of generation one.[2]

The program in Figure 6 expects as input a single node labelled with the generation number of the Sierpinski triangle to be produced. The rule `init` creates the Sierpinski triangle of generation 0 and turns the input node into a "control node" with label x:0, holding the required generation number x together with the current generation number.

After initialisation, the nested loop (`inc; expand!)!` is executed. In each iteration of the outer loop, `inc` increases the current generation number if it is smaller than the required number (which is checked by the rule's condition). If the test is successful, the inner loop `expand!` performs a Sierpinski step on each triangle whose top node is labelled with the current generation number: the triangle is replaced by four triangles such that the top nodes of the three outer triangles are labelled with the next higher generation number. The test $x > y$ fails when the required generation number has been reached. In this case the application of `inc` fails, causing the outer loop to terminate and return the current graph which is the Sierpinski triangle of the requested generation.

Sierpinski triangles pose a hard challenge for graph transformation: generating the n-th triangle requires space and a number of rule applications exponential in n. This problem was part of the 2007 tool contest for graph transformation, where the goal was to generate triangles of generation numbers as high as possible and as fast as possible [19].

4 Reference Interpreters: Uses and Requirements

A reference interpreter for a new programming language such as GP2 has several potential uses. Each has consequences for the way the reference interpreter is written and the facilities it provides.

An arbiter for programmers. A programmer working in a new language needs to know whether what they are writing is a valid program, and whether the effect of executing it is the effect they intend. To resolve such issues, the programmer may want to use a reference interpreter as a black box, checking the output it produces given their program as input. Or they may wish to look at a salient part of the source-code for the interpreter, to confirm some aspect of the language they are unsure about.

It follows that a reference interpreter should provide as output at least a report whether a program is valid, and if so a clear representation of the result when it is evaluated. It also follows that the source-code for a reference interpreter should be organised in such a way that salient components are easy to identify. For ease of reading it should be written using a consistent style in a modest subset of a suitable high-level language.

An arbiter for implementors. An implementer of a programming language, developing their own interpreter or compiler, needs a standard against which to test the correctness of their implementation. There are two main respects in which any implementation should agree with a reference interpreter as a defining standard. They should agree which programs are valid, and for valid programs they should agree the results of executing them. Like application programmers, implementers too may wish sometimes to use the reference interpreter as a black box, but at other times to consult its internal definitions.

There are additional requirements for this use, bearing in mind the likely development or generation of many test programs. The representation of the reference interpreter's results for such programs should be amenable to automated comparison. This comparison presents particular challenges in GP 2 since behaviour of programs may be non-deterministic, or programs may not terminate, or both. The number of test programs may be large — there may even be arbitrarily many test programs generated dynamically.
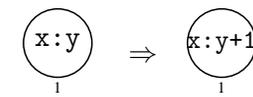
---

[2]The geometric layout was created by the graphical interface of the GP 1 implementation [13].

```
Main = init; (inc; expand!)!
```

init(x : int)
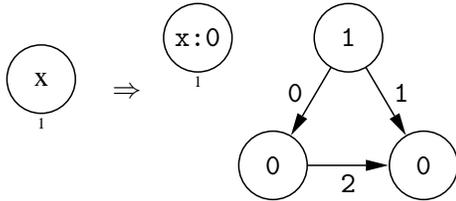
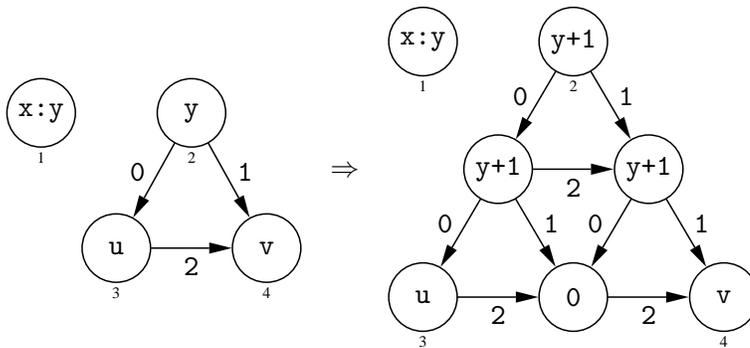inc(x, y : int)

where x > y

expand(x, y, u, v : int)

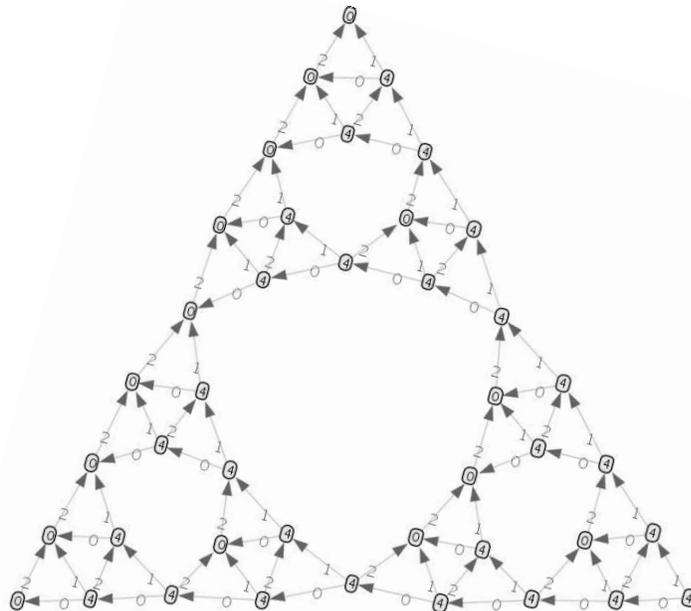Figure 6: Program for generating Sierpinski triangles

Figure 7: Third generation Sierpinski triangle

So although performance is not a design goal for the reference interpreter, its performance should be good enough to make such multi-test comparisons feasible.

A prototype for application developers. If no production compiler has been developed for the language, or none is yet available to an application developer, they may need to use a reference interpreter as an initial development platform.

During the development of application programs, errors are common. So, for this use, a reference interpreter should provide not only a check for valid programs, but a rapid check with informative reports of errors. Yet elaborate error handling must not obscure the definitional style in which the interpreter is written. Similarly, it is desirable to have the option of some kind of trace or other informative report to shed light on failures or unexpected results when a program is evaluated. Here again, the machinery must not obscure the basic definitions for evaluation, nor should it impose heavy performance costs when performance of the interpreter has already been sacrificed in favour of simplicity.

A prototype for implementation developers. As well as using a reference interpreter to verify correctness, implementers may wish to use it as the starting point in the development of another interpreter or a compiler. The whole course of such a development might even be defined as the successive replacement of interpreter components by alternatives giving higher performance, or richer information, at the cost of greater complexity. The advantage of this approach is that as each replacement is introduced it can be checked as a new component in an already tried system.

This use of a reference interpreter requires a modular design with simple and clearly defined interfaces between components. Concerns should be separated so far as possible, avoiding dependencies that are not strictly necessary. Options for development by successive replacement may be further increased by choosing a host programming system for the reference interpreter that has a well-developed foreign-language interface.

5 Implementation

We describe the key components of the reference interpreter with the aim of illustrating the simplicity, clarity, and conciseness of the implementation. A basic knowledge of Haskell is useful but not essential to understand the content in the following sections.
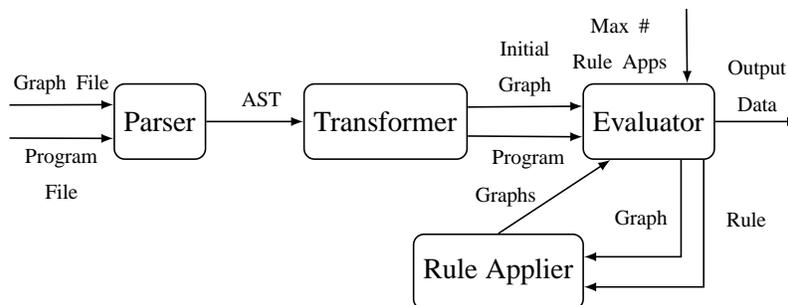

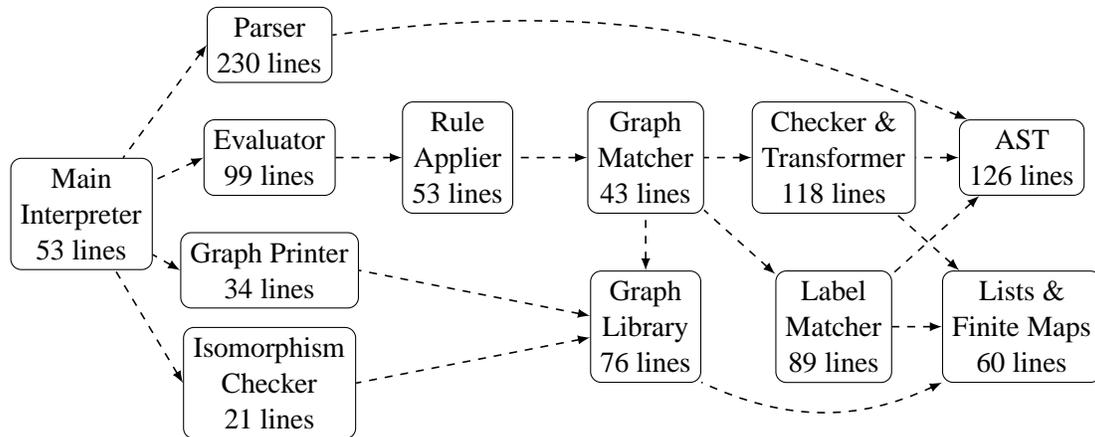Figure 8: Main data flow of the reference interpreter

Figure 9: Module dependencies. A module points to any modules on which it depends. Line counts exclude blank lines and comment-only lines

5.1 Overview

Figure 8 shows a data flowchart of the reference interpreter. It takes three inputs: (1) a file containing the textual representation of a GP 2 program, (2) a file containing the textual representation of a host graph, and (3) an upper limit on the number of rule applications to be made before halting program execution. It runs the program on the host graph, traversing either all nondeterministic branches of the program or a single branch, at the behest of the user. The output data is a complete description of all possible outputs. Section 5.7 describes the output data in detail.

The interpreter contains approximately 1,000 lines of Haskell source code. Figure 9 shows the module dependency structure of the interpreter and an indication of module sizes.

5.2 Parser

The parser has two components: (1) a host graph parser and (2) a program text parser. Each individual parsing function takes a string as input and attempts to match a prefix of the string to a particular syntactic unit. It uses a library of parser combinators. Their purpose is to neatly compose the parsing functions to cover standard parsing requirements such as alternation and repetition. The parsing code is very similar in appearance to GP 2's context-free grammar: each nonterminal of the grammar is represented by a Haskell function that parses the right-hand side of the grammar rule. For example:

```
gpMain :: Parser Main
gpMain = keyword "Main" |> keyword "=" |> pure Main <*>
         commandSequence
```

The operators `|>` and `<*>` are binary functions: `|>` ignores the output of its left parser and `<*>` sequences two parsers. Applications of `keyword` recognise and discard a string argument, and `commandSequence` is another parsing function. `Main` is a data constructor for the main node of GP 2's abstract syntax tree.

5.3   Checking & Transformation

The checking and transformation phase extracts semantic information from the AST, such as the types of variables specified in a rule schema's parameter list, and transforms both rule graphs and the host graph into the data structure defined in the graph library. The internal graph representation is a pair of maps from keys to labels for each of nodes and edges separately. Node keys are integers. Edge keys are triples: source key, target key and an integer. Node and edge labels are encoded into the node and edge data types. Operations on graphs are concisely represented using Haskell functions from the Haskell library `Data.Map` which implements maps efficiently as balanced binary trees. Node and edge enumeration functions also support the use of Haskell's strong list-processing. See Section 5.5 for details.

5.4   Label Matching

The label matching algorithm establishes whether a label from a rule's left-hand side can be matched with a label from the host graph. It takes as input the current *environment*, the set of bindings for label variables, and the two labels to be compared.

   GP 2 labels consist of a mark and a list. The marks are encoded as an abstract data type and are directly comparable. GP 2's lists are naturally encoded as Haskell lists, where each element is a GP 2 atom. Atoms occurring in the host graph are constants (integers, characters or strings), while rule atoms are either constants, variables or a concatenated string[3]. If a match binds a variable, the binding must define a compatible extension of the environment.

   When comparing atoms, the interesting case occurs if a list variable is encountered. GP 2 allows at most one list variable in any label expression on a left-hand side. This restriction allows binding to host-label segments of determined length, by comparing the lengths of the remainder of the rule label and the remainder of the host label. Matching fails if too few host atoms remain.

5.5   Graph Matching

Given a rule graph L and a host graph G, the graph matcher lazily constructs a list of `GraphMorphisms`. A `GraphMorphism` is a data structure containing an environment, a mapping between nodes in L and the corresponding nodes in G, and a similar edge map. We use association lists to represent these small mappings, for simplicity and amenability to list-processing. Morphisms are generated in two stages. First the candidate `NodeMorphisms` are identified, where a `NodeMorphism` is an environment and a node mapping. For each such `NodeMorphism`, the matcher searches for compatible edge mappings and environment extensions to form a set of complete `GraphMorphisms`.

Node matching. For each node $l_k \in L$, the matcher constructs the list of all host nodes $[h_{k_1}, \ldots, h_{k_m}]$ that match l_k with respect to label matching and rootedness[4] An environment is paired with each host node. The result is a list of lists $[[h_{1_1}, \ldots, h_{1_m}], \ldots, [h_{n_1}, \ldots, h_{n_m}]]$ where n is the number of nodes in L. A candidate node mapping is found by injectively selecting one item from each list. The final step is to test each candidate mapping for compatibility with respect to its environment. Haskell's list comprehensions are perfectly suited for this task: the list of lists is computed with a single nested list comprehension, while a second list comprehension is responsible for collating the valid candidate mappings.

---

   [3]Expressions and degree operators are forbidden in LHS labels to prevent ambiguous matching.

   [4]Graphs can be augmented with root nodes to reduce the search space. GP 2's semantics requires that a root node in L must only match a root node in G [2].

Edge matching. For each edge in *L*, we use a candidate node morphism to determine the required source and target for a corresponding edge in the host graph. The list of candidate host edges is the list of host edges from that source to that target. Each rule edge is checked against each candidate host edge for label compatibility, supported by the environment passed from the node morphism.

5.6 Rule Application

Each of the `GraphMorphisms` produced by the graph matcher is checked against a *dangling condition* and any rule conditions. If these checks succeed, the rule application is performed in the following steps: delete edges, delete nodes, relabel nodes, add nodes, relabel edges, add edges. For relabelling, variables take their values from a `GraphMorphism`'s environment.

The dangling condition can be elegantly expressed as follows.

```
danglingCondition :: HostGraph -> EdgeMatches -> [NodeId] -> Bool
danglingCondition h ems delns =
        null [e | hn <- delns, e <- incidentEdges h hn \\ rng ems]
```

The second argument is an edge map, obtained from a `GraphMorphism`. The third argument is the set of nodes deleted by the rule. The function body specifies that no host edge `e` incident to any deleted node `n` may lie outside of the range of the edge map `ems`.

5.7 The Evaluator

The evaluator applies a GP 2 program to a host graph, subject to an upper bound on the number of rule applications. Often the same graph can be reached through several distinct computational branches. Therefore, when program execution is complete, an isomorphism checker is used to collate the list of output graphs into its isomorphism classes. The output is as follows:

1. A list of unique output graphs, up to isomorphism, with a count of how many isomorphic copies of each graph were generated.

2. The number of failures. For example, a failure occurs in some contexts if none of a set of rules can be applied to a graph.

3. The number of unfinished computations. A computation is unfinished if the bound on rule applications is reached before the end of the main command sequence.

During program execution the evaluator maintains a list of `GraphStates`, one for each nondeterministic branch of the computation so far. A `GraphState` is one of: (1) a graph with its rule application count, (2) a failure symbol with its rule application count, and (3) an unfinished symbol. Each GP 2 control construct is evaluated by a function that takes as input a single `GraphState` and some program data, returning a list of `GraphStates`. Only the application of a rule can yield a `GraphState` with a changed graph. The rule application process is the workhorse of the interpreter, so here by way of illustration is the top-level defining equation for the evaluation of a rule-call command:

```
evalSimpleCommand max ds (RuleCall rs) (GS g rc) =
   if rc == max then [Unfinished]
   else case [h | r <- rs, h <- applyRule g $ ruleLookup r ds] of
        [] -> [Failure rc]
        hs -> [GS h (rc+1) | h <- hs]
```

Here `max` is the rule application bound, `ds` is a list of the rule and procedure declarations in the GP 2 program, `rs` is a list of rules, and `GS g rc` is the current graph state. `GS` is the `GraphState` constructor, `g` is the working host graph, and `rc` is the number of rules that have been applied to `g`. The case-subject list comprehension can be read as, "for all rules `r` in `rs`, apply `r` to `g` and produce the list of all output graphs `h`." Each individual rule application may produce multiple output graphs; the list comprehension gathers every possible output into a single lazily-computed list. If `resultGraphs` is empty, then no rule in `rs` was applicable, and the list containing the single `GraphState Failure` is returned. Otherwise, the output graphs are placed into a fresh list of `GraphStates`, each with an incremented rule-application count.

6 Performance Evaluation

In this section we will look at how efficiently our interpreter executes the benchmark programs described in Section 3, and discuss the factors that affect its performance. Though not tuned for speed, the interpreter must run fast enough to allow its use as a practical tool.

6.1 The Test Environment

We compiled the interpreter using the Glasgow Haskell Compiler [1] version 7.6.3 with optimisations and profiling support enabled:

```
$ ghc -O2 -prof -fprof-auto -rtsopts -o gp2 Main.hs
```

All figures reported were obtained using a quad-core Intel i7 clocked at 3.4GHz, with 8GB RAM, running 64-bit Ubuntu 14.04 LTS with kernel 3.13.0. The number of processor cores should not have a significant effect on the measured performance of the single-threaded GP 2 interpreter.

We ran benchmarks using the following command

```
$ timeout --foreground 5m time \
      gp2 +RTS -p -sgc.prof -RTS $GPOPT $PROG $GRAPH 10000
```

limiting execution time to five minutes for each application of a program to a host graph. We used the sum of user and system time reported by the standard `time` utility as our measure of execution time. The arguments to `gp2` between `+RTS` and `-RTS` tell the Haskell run-time system to save profiling information. The `$GPOPT` variable was either set to `--one` to put the interpreter into single-result mode (see Table 1), or unset for all-result mode (see Table 2). The final three mandatory arguments to the `gp2` executable specify the benchmark program, the host graph, and the maximum number of rule applications, as described in Section 5.

6.2 Host Graphs

The names of host graphs used for benchmarking give an indication of their structure.

Gen n. The *Sierpinski* program expects a host graph containing a single node with a numeric label, which controls the number of iterations of the `expand!` command.

Linear n. A chain of *n* nodes. The first node has only a single outgoing edge. The last node has only a single incoming edge. All other nodes have exactly one incoming and one outgoing edge.

Cyclic n. As Linear *n*, but with an extra edge from the last node to the first, so every node has exactly one incoming and one outgoing edge.

x × y Grid. A rectangular lattice *x* nodes wide by *y* nodes tall, with $x(y-1) + y(x-1)$ edges. The *shortest distances* benchmark requires all edges to have an integer "cost" of traversal. The *grid* host graphs passed to this program have the top-left node marked grey, all edges directed either rightwards or downwards, a cost of one assigned to half of the edges, and a cost of two to the other half.

6.3 Benchmark performance

Single-result mode. Table 1 summarises results for the reference interpreter operating in single-result mode. The *Apps* column shows the number of rule applications required to reach the solution. *Time* lists the sum of user and system time reported by the `time` command. The final two columns show the maximum amount of memory requested by the gp2 executable, and the maximum memory holding live data respectively. The disparity between these two numbers, which sometimes approaches a factor of three, results from the Haskell run-time system requesting memory from the operating system in large chunks.

All-result mode. Table 2 summarises the performance of the reference interpreter running in all-result mode. This table contains three additional columns showing the total number of output graphs, the number of distinct output graphs up to isomorphism, and the number of executions that terminated in failure. Where different solutions required differing numbers of rule applications the *Apps* column now shows the range of values.

The extra costs of evaluating a program in all-result mode go beyond those of generating all possible output graphs; the interpreter must also test them for isomorphism. Unsurprisingly, execution time increases sharply with increasing size of host graph, putting many of the computations that completed in single-result mode beyond our five-minute execution-time limit.

The effect on heap usage of producing all possible results is less than one might expect for the *3x3 grid* host graph in both the *acyclicity test* and *shortest distances* programs, given the tens of thousands of isomorphic graphs generated. We benefit from Haskell's lazy evaluation of the list of output graphs. As there is a single isomorphism class, at most two final host graphs are needed in memory simultaneously — though there may be many intermediate graphs awaiting further processing.

In contrast, the *vertex colouring* benchmark has many distinct solutions. As the five minute limit approached during all-results computation for the *3x3 grid* host graph, gp2 had been allocated over seven gigabytes, putting a conservative estimate of its live heap in excess of two gigabytes!

6.4 Discussion

In single-result mode, performance is acceptable even for some quite complex programs. However, in all-result mode, execution time and memory usage can increase very rapidly with problem size. An extreme example is the vertex-colouring program, which exhibits factorial growth in the number of possible intermediate graphs as edge-counts in initial graphs increase.

The current version of the interpreter uses a finite-map library for indexed sets of nodes and edges in graphs. Early versions stored these sets as association lists, resulting in an interpreter which spent most of its execution time traversing lists of nodes and edges. The cumulative effect of several incremental improvements to our original prototype, without making it larger or more complicated, was a large speed-up. This in turn enabled us to run larger computations, putting greater stress on stack and heap

| Benchmark | Host Graph | Apps | Time/s | Heap/kB | |
				Allocd	Live
Acyclicity test	3x3 grid	12	0.02	2048	129
	5x5 grid	40	0.03	3072	382
	7x7 grid	84	0.17	4096	1119
	9x9 grid	144	0.70	6144	2100
	cyclic 100	0	0.04	3072	778
	cyclic 500	0	0.46	14336	5646
	cyclic 1000	0	1.76	25600	10368
Shortest distances	5x5 grid	38	< 0.01	3072	414
	7x7 grid	90	0.08	4096	1177
	9x9 grid	175	0.39	8192	3172
Sierpinski	gen 2	7	< 0.01	2048	133
	gen 3	17	0.14	5120	1056
	gen 4	45	6.52	58368	18313
	gen 5	-	> 5m	-	-
Transitive closure	linear 05	6	< 0.01	2048	144
	linear 10	36	0.04	2048	144
	linear 20	171	1.67	21504	7073
	linear 30	406	14.39	103424	33152
	linear 40	741	66.31	324608	103275
	linear 50	-	> 5m	-	-
Vertex colouring	3x3 grid	27	0.02	2048	140
	5x5 grid	125	0.03	3072	999
	7x7 grid	343	0.17	9216	3681
	9x9 grid	729	0.89	25600	11438

Table 1: Reference interpreter benchmark results when generating a single output graph

memory. There may yet be quite simple modifications that would reduce memory demand — we have made comparatively little effort in this direction.

As discussed in Section 5.5 the reference interpreter matches nodes and edges in separate passes. This makes for a simple algorithm at the expense of performance. A more performance focussed implementation might use a *search plan* [4, 21] in which a graph morphism is built incrementally by adding both nodes and edges to an existing partial morphism, back-tracking if no suitable candidate can be found.

7   Related and Future Work

Early programming languages were often defined by their implementations, perhaps in the form of a *definitional interpreter*. We now have more abstract techniques for defining operational semantics. However, in recent years there has been a rehabilitation of interpreters as executable counterparts to semantic definitions — eg. [3]. Motivation varies, but here's an extract from the preface of an influential textbook:

Benchmark	Host Graph	Output Graphs					Heap/kB	
		Total	Unique	Failed	Apps	Time/s	Total	Live
Acyclicity test	2x2 grid	6	1	0	4	< 0.01	2048	134
	3x3 grid	19770	1	0	12	12.00	10240	3301
	4x4 grid	-	-	-	-	> 5*m*	-	-
	cyclic 100	0	0	100	0	0.06	4096	784
	cyclic 500	0	0	500	0	0.86	14336	5651
	cyclic 1000	0	0	1000	0	3.31	26624	11053
Shortest distances	2x2 grid	6	1	0	4	< 0.01	2048	131
	3x3 grid	28924	1	0	9-14	19.15	167936	58180
	4x4 grid	-	-	-	-	> 5*m*	-	-
Sierpinski	gen 2	6	1	0	7	0.04	3072	242
	gen 3	-	-	-	-	> 5*m*	-	-
Transitive closure	linear 05	866	1	0	6	0.44	6144	1699
	linear 10	-	-	-	-	> 5*m*	-	-
Vertex colouring	2x2 grid	480	2	0	6-8	0.07	5120	1598
	3x3 grid	-	-	-	-	> 5*m*	-	-

Table 2: Reference interpreter benchmark results when generating all possible output graphs

> *Our goal is to provide a deep, working understanding of the essential concepts of programming languages. . . . Most of these essentials relate to the semantics, or meaning, of program elements. Such meanings reflect how program elements are interpreted as the program executes. . . . The most interesting question about a program is,* "What does it do?" *The study of interpreters tells us this. Interpreters are critical because they reveal nuances of meaning, and are the direct path to more efficient compilation and to other kinds of program analyses.* [8]

In several respects, our motivation is similar. We adopt the slogan: *Semantics first!*. But then, following the semantic definition, we write a reference interpreter in order to promote a "*deep, working understanding*" of the GP 2 design, and to find "*path(s) to more efficient compilation . . . and program analysis*".

Languages based on graph-transformation rules include PROGRES [18], AGG [5, 17], GAMMA [7], GROOVE [9], GRGEN.NET [10] and PORGY [6]. To our knowledge, none of these languages has a published implementation in the same spirit as our reference interpreter. For example, GROOVE and GRGEN.NET are two of the most widely used systems. The Java source code for the GROOVE implementation, including a graphical development suite, extends to around 150,000 lines. GRGEN.NET is implemented in a combination of Java and C#: a Java front-end is used to generate C# code and .NET assemblies from a textual specification of a GRGEN program; the run-time system and other components are written in C#. In all there are around 68,000 lines of Java source for the front-end, and around 93,000 lines of C# for the run-time system, API support and an interactive shell. We recognise that both GROOVE and GRGEN.NET are mature and fully-featured systems, and GRGEN.NET in particular is highly optimising. Even so, the contrast with the 1,000-line Haskell sources for our GP 2 reference interpreter is striking.

We have begun work on two compiled implementations of GP 2. One generates code for an abstract machine; the other translates GP 2 programs to C. They also differ in the way a low-level graph data

structure is defined and accessed, and the strategies employed to match left-hand sides of rules. The reference interpreter is supporting these ongoing developments. For example, some front-end components are re-used, and we check output graphs against isomorphism classes computed by the interpreter.

8   Conclusions

Our original goals for our reference interpreter have largely been realised. We have a concise implementation of GP 2, expressed in around 1,000 lines using the lazy functional language Haskell. We have taken every opportunity to use a Haskell strength — lazy list-processing, and in particular list comprehensions for generate-and-test style definitions — to achieve this conciseness. However, despite our observations in Section 4 about error reports and traces, we concede that our current interpreter provides only a bare minimum in this respect.

As stated in the Introduction, our motivation for producing a simple interpreter was to achieve clarity and correctness. This raises the question of whether the reference interpreter could be formally verified against the operational semantics of GP 2. While this is a desirable goal for future work, existing verification projects for subsets of C [12] and ML [11] indicate that such a project would be a major endeavour despite the modest size of the GP 2 language.

When working with the interpreter, we have had some unexpected results. Occasionally, the practical consequences of a crisp semantic definition may be surprising to programmers, or it may pose challenges for an efficient implementation. We have found that our reference interpreter can shed helpful light in such instances.

As we have shown in Section 6, the interpreter is efficient enough for practical use in testing, both by GP 2 programmers and by the developers of other GP 2 implementations. Our main reservation here concerns all-results mode. Used in this mode, the interpreter can require very long execution times and all the memory our machines have available. One remedy might be to check for isomorphism or other equivalences between intermediate graphs, compacting the state-space. However, the extra machinery would complicate the interpreter, and it could demand even more space in some cases. Instead, our likely solution will be to build up a standard set of test programs. We can first run each test (for several days, if necessary) on a powerful machine to produce the set of all possible output graphs up to isomorphism. Our isomorphism checker, though simple, is efficient enough for rapid subsequent checking of single results produced by another implementation.

References

[1] *The Glasgow Haskell Compiler*. `http://www.haskell.org/ghc/`. Accessed 19 January 2015.

[2] Christopher Bak & Detlef Plump (2012): *Rooted Graph Programs*. In: *Proc. International Workshop on Graph Based Tools (GraBaTs 2012)*, *Electronic Communications of the EASST* 54.

[3] Brian Campbell (2012): *An Executable Semantics for CompCert C*. In: *Proc. International Conference on Certified Programs and Proofs (CPP 2012)*, *Lecture Notes in Computer Science* 7679, Springer, pp. 60–75, doi:`10.1007/978-3-642-35308-6_8`.

[4] Heiko Dörr (1995): *Efficient Graph Rewriting and its Implementation*. *Lecture Notes in Computer Science* 922, Springer, doi:`10.1007/BFb0031909`.

[5] Claudia Ermel, Michael Rudolf & Gabi Taentzer (1999): *The AGG Approach: Language and Environment*. In H. Ehrig, G. Engels, H.-J. Kreowski & G. Rozenberg, editors: *Handbook of Graph Grammars and Computing by Graph Transformation*, chapter 14, 2, World Scientific, pp. 551–603.

[6] Maribel Fernández, Hélène Kirchner, Ian Mackie & Bruno Pinaud (2014): *Visual Modelling of Complex Systems: Towards an Abstract Machine for PORGY*. In: *Proc. Computability in Europe (CiE 2014)*, Lecture Notes in Computer Science 8493, Springer, pp. 183–193, doi:10.1007/978-3-319-08019-2_19.

[7] Pascal Fradet & Daniel Le Métayer (1998): *Structured Gamma*. Science of Computer Programming 31(2–3), pp. 263–289, doi:10.1016/S0167-6423(97)00023-3.

[8] Daniel P. Friedman & Mitchell Wand (2008): *Essentials of Programming Languages*, 3rd edition. The MIT Press.

[9] Amir Hossein Ghamarian, Maarten de Mol, Arend Rensink, Eduardo Zambon & Maria Zimakova (2012): *Modelling and analysis using GROOVE*. International Journal on Software Tools for Technology Transfer 14(1), pp. 15–40, doi:10.1007/s10009-011-0186-x.

[10] Edgar Jakumeit, Sebastian Buchwald & Moritz Kroll (2010): *GrGen.NET - The expressive, convenient and fast graph rewrite system*. International Journal on Software Tools for Technology Transfer 12(3–4), pp. 263–271, doi:10.1007/s10009-010-0148-8.

[11] Ramana Kumar, Magnus O. Myreen, Michael Norrish & Scott Owens (2014): *CakeML: a verified implementation of ML*. In: *Proc. Symposium on Principles of Programming Languages (POPL 2014)*, ACM Press, pp. 179–192, doi:10.1145/2535838.2535841.

[12] Xavier Leroy (2009): *Formal verification of a realistic compiler*. Communications of the ACM 52(7), pp. 107–115, doi:10.1145/1538788.1538814.

[13] Greg Manning & Detlef Plump (2008): *The GP Programming System*. In: *Proc. Graph Transformation and Visual Modelling Techniques (GT-VMT 2008)*, Electronic Communications of the EASST 10.

[14] Detlef Plump (2009): *The Graph Programming Language GP*. In: *Proc. International Conference on Algebraic Informatics (CAI 2009)*, Lecture Notes in Computer Science 5725, Springer, pp. 99–122, doi:10.1007/978-3-642-03564-7_6.

[15] Detlef Plump (2012): *The Design of GP 2*. In: *Proc. Workshop on Reduction Strategies in Rewriting and Programming (WRS 2011)*, Electronic Proceedings in Theoretical Computer Science 82, pp. 1–16, doi:10.4204/EPTCS.82.1.

[16] Christopher M. Poskitt & Detlef Plump (2012): *Hoare-Style Verification of Graph Programs*. Fundamenta Informaticae 118(1-2), pp. 135–175, doi:10.3233/FI-2012-708.

[17] Olga Runge, Claudia Ermel & Gabriele Taentzer (2012): *AGG 2.0 — New Features for Specifying and Analyzing Algebraic Graph Transformations*. In: *Proc. Applications of Graph Transformations with Industrial Relevance (AGTIVE 2011)*, Lecture Notes in Computer Science 7233, Springer, pp. 81–88, doi:10.1007/978-3-642-34176-2_8.

[18] Andy Schürr, Andreas Winter & Albert Zündorf (1999): *The PROGRES Approach: Language and Environment*. In H. Ehrig, G. Engels, H.-J. Kreowski & G. Rozenberg, editors: *Handbook of Graph Grammars and Computing by Graph Transformation*, chapter 13, 2, World Scientific, pp. 487–550.

[19] Gabriele Taentzer, Enrico Biermann, Dénes Bisztray, Bernd Bohnet, Iovka Boneva, Artur Boronat, Leif Geiger, Rubino Geiß, Ákos Horvath, Ole Kniemeyer, Tom Mens, Benjamin Ness, Detlef Plump & Tamás Vajk (2008): *Generation of Sierpinski Triangles: A Case Study for Graph Transformation Tools*. In: *Applications of Graph Transformation with Industrial Relevance (AGTIVE 2007)*, Revised Selected and Invited Papers, Lecture Notes in Computer Science 5088, Springer, pp. 514–539, doi:10.1007/978-3-540-89020-1_35.

[20] Gergely Varró, Andy Schürr & Dániel Varró (2005): *Benchmarking for Graph Transformation*. In: *Proc. IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC 2005)*, IEEE Computer Society, pp. 79–88, doi:10.1109/VLHCC.2005.23.

[21] Albert Zündorf (1996): *Graph Pattern Matching in PROGRES*. In: *Proc. Graph Grammars and Their Application to Computer Science, Lecture Notes in Computer Science* 1073, Springer, pp. 454–468, doi:10 . 1007/3-540-61228-9_105.