Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 211 (2008) 231–240
www.elsevier.com/locate/entcs

The York Abstract Machine

Greg Manning[1] Detlef Plump[2]

Department of Computer Science
The University of York, UK

Abstract

We introduce the York Abstract Machine (YAM) for implementing the graph programming language GP and, potentially, other graph transformation languages. The advantages of an abstract machine over a direct interpreter for graph transformation rules are better efficiency, use as a common target for compiling both future versions of GP and other languages, and portability of GP programs to different platforms.

Keywords: Graph transformation; GP; abstract machines; nondeterminism; backtracking

1 Introduction

The graph programming language GP [6] consists in its core of just three constructs: application of a set of conditional rule schemata either (1) in a single step or (2) as long as possible, and (3) sequential composition of programs. This language is computationally complete (see [5]) and has a simple formal semantics. In this paper, we present a low-level abstract machine for graph transformation—the York Abstract Machine (YAM)—that will be used to implement GP.

A major advantage of a low-level abstract machine over a high-level interpreter for graph transformation rules is higher speed. The instructions of the abstract machine are typically simple stack operations which, after a GP program has been compiled, need not analyse the left- and right-hand graphs of a rule over and over again. Instead, the analysis of rules is performed once and for all when GP programs are translated into YAM bytecode.

The YAM can also serve as a common target for compilers of both future versions of GP—a language still under development—and, potentially, other graph transformation languages. Moreover, the YAM will support the portability of GP

[1] Email: gm@cs.york.ac.uk
[2] Email: det@cs.york.ac.uk

doi:10.1016/j.entcs.2008.04.045

programs because they can be compiled to bytecode with any available compiler and then executed on every platform on which a YAM implementation exists.

To give an example of YAM code, consider the GP program `VertexColouring` of Figure 1. The program labels all nodes of an input graph with colours (integers) such that any adjacent nodes have different colours. To achieve this, first the rule `Init` is applied as long as possible in order to label all nodes with colour 1. Then the `Inc`-rules nondeterministically increment colours until any adjacent nodes are differently coloured.

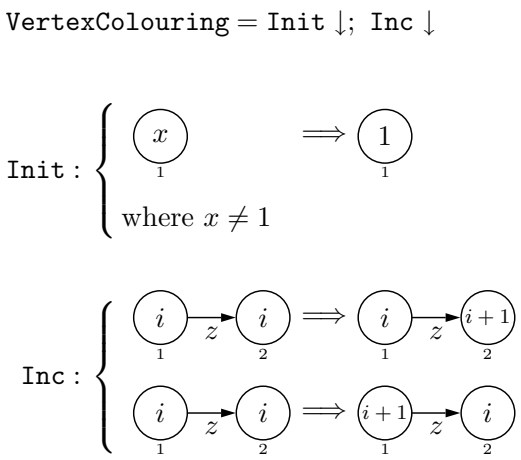


Fig. 1. GP program `VertexColouring`

A full version of (hand-compiled) YAM code for this program is presented in the appendix. Here we only consider the code for the conditional rule `Init`:

```
NA
Dup Get_node_label ATOI 1 Equals Not Assert
"1" Relabel_node
```

The purpose of these instructions can be described as follows: "Find any node, check its label is not 1, and relabel it to 1".

The first instruction, `NA`, pushes a node identifier onto the stack. If some later instruction fails and the machine starts backtracking, this instruction will subsequently retry with the next node (for some, usually random, ordering of nodes) until all nodes have been tried. The sequence from `Dup` to `Assert` checks that the label is not 1, in the following manner: `Dup` duplicates the top of stack;[3] `Get_node_label` pops a node identifier from the top of stack and pushes the label of that node (a string); `ATOI` converts the top of stack from a string to an integer; `1` pushes the integer value 1 onto the stack; `Equals` pops two values from the stack and pushes 1 if they are equal, 0 otherwise; `Not` pops the top of stack and pushes 1 if it is 0,

[3] Since stack instructions are almost always destructive, it is often necessary to save values by duplicating them.

0 otherwise; and `Assert` pops the top of stack and starts backtracking if it is 0. The final sequence, `"1" Relabel_node`, relabels the node with the string "1". (The identifier of the node to relabel is still on the stack, thanks to the earlier `Dup`.)

The next section describes the YAM language and its execution. Section 3 further discusses the compilation of GP to YAM Code. In Section 4 we explain the graph data-structure used to implement the YAM, Section 5 focuses on the three stacks the machine is based on. Section 6 briefly addresses the relation of the YAM to graph transformation languages other than GP, and Section 7 gives some concluding remarks and a few topics for future work.

2 The YAM language

A program in the YAM language (the assembly) is, at its most basic, a sequence of instructions separated by whitespace. It is also possible to use labels and macros, as discussed below.

A *program label* is declared in the form `LabelName:` — the name of the label followed by a colon. Whenever it is found in the source code, the integer value of the program counter at the definition is pushed onto the data stack. This value is typically then used to jump control to that program counter (via a `Jump`, `ALAP` or `Choice` instruction).

YAM code can also contain *macros* which are simply named sequences of instructions. An example of a macro definition can be found in the code for `Vertex-Colouring`:

```
:EXXA NA Dup Get_node_label Swap EILA ;.
```

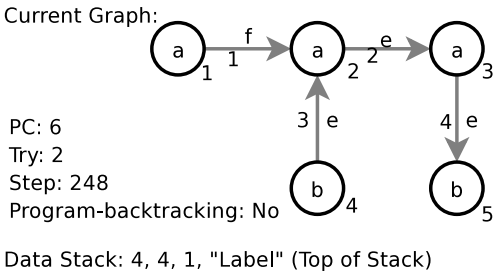
This is a macro named `EXXA` which finds an edge between two nodes with the same label as follows: Find any node, find its label, then find an edge from that node to a node with that label. Whenever a macro call is found in the code, it is replaced verbatim with the body of the macro (macro expansion). The *prelude* is a useful collection of macros.

The YAM instructions alter the state of the machine, which consists of a single graph, three stacks—data stack, choice stack and graph change stack, several integer values—current program counter (PC), current step, current try [4] —and a flag to control the behaviour of backtracking (explained later in this section). For example, Figure 2 shows a YAM state in the middle of a computation. There have been two changes to the graph since the machine started, represented by two frames on the graph change stack. Also, there have been two decisions made, represented by the two frames on the choice stack. The top two elements on the stack are the string "Label" and the integer 1. The next instruction for this machine might be to relabel node 1 from 'a' to 'Label'.

The graphs which the YAM uses are directed, labelled graphs. The labels of edges and nodes are strings. Parallel edges are permitted, as are self loops.

There are currently three types of data that can go on the data stack: strings,

[4] The try number is the number of times the current instruction has been attempted.



Graph Change Stack: (Top of Stack)	Choice Stack: (Top of Stack)
PC:3,Step 61, Added edge 4.	PC:2,Step 60, try 1, Data Stack: 4
PC:0, Step 0, Added node 5.	PC:1,Step 59, try 3, Data Stack:

Fig. 2. An example of a YAM state

integers and boolean values. Booleans are represented by integers: non-zero integers represent logical 'true' and zero represents 'false'. Strings can be converted to integers and integers to strings using the instructions `ATOI` and `ITOA`, respectively.

There are currently about 70 instructions which the YAM can execute.[5] They fall into four main categories:

(i) Data stack-only instructions such as `Add`, `Multiply`, `Swap` and `Duplicate`.

(ii) Control instructions, which delimit rules, explicitly represent choices or mark deterministic sections of code, such as `ALAP`, `Once` and `Cut`.

(iii) Graph query instructions such as `NA`, `EILL` and `Get_edge_start`.

(iv) Graph modification instructions such as `Add_node` and `Del_edge`.

Data stack-only instructions manipulate the data stack in a simple way, having no effect on the other stacks or the current graph. Control instructions influence the choice stack or the program counter. Graph query instructions modify the choice stack where they have returned one of many answers, and push their answer onto the top of the data stack. Graph modification instructions modify the graph change stack (recording data so that the graph change can be undone) and the current graph.

The YAM provides backtracking to implement the nondeterminism of graph transformation programs. Certain instructions involve the machine choosing which answer of several to return. If the machine gets the answer wrong in that some later part of the program fails, then the choice needs to be revisited and a different answer chosen. Stacks are used to remember these choices so that they can later be reconsidered.

There are two different types of backtracking. *Matching backtracking* is concerned with making the right choices of nodes and edges to find an individual match for a rule. *Program backtracking* deals with choosing the right rule to apply from a set, and determining when an as-long-as-possible operator has finished. The program-backtracking boolean flag in the YAM state is to record which type of backtracking is currently running.

The YAM currently has two modes of operation: *one-result* and *all-results*. In

[5] The full list of YAM instructions is available at
http://www-users.cs.york.ac.uk/~gm/YAM/instructions.html

either case, the machine runs until the first result is found. If it is in one-result mode then it terminates, if it is in all-results mode then it imposes a failure and finds the next result. It continues in this way until it finally backtracks past the first choice. In such a manner, the machine will find all results in the search space, so long as there are no infinite computation paths. If there are infinite computation paths then the machine will enter such a path at some point, possibly before producing all results or any result at all.

3 Compiling GP to YAM code

Although a compiler is still under development, we can make a few general remarks about the process of compiling GP to YAM code. First, individual rules need to be translated into fragments of YAM code. These fragments can then be put together with appropriate control instructions between them to mark them as sets of rules and iterated or single-step applications.

Compiled GP programs are executed using a depth-first strategy. In order to compute a result, rules must be matched and applied until the end of the program is reached. To apply each rule, an instance of the left-hand side must be found in the current graph (this is the subgraph isomorphism problem), and any conditions of the rule must be checked. At GP to YAM compile time, this problem is broken down into smaller problems of finding individual nodes and edges with certain characteristics (such as "a node with label l" or "an edge from node 3"), and asserting conditions (such as "the label is not 1"). The correct choices (if any exist) for each of these small decisions will lead to a result for the program and current input graph. The correct choices are found by means of backtracking, in a manner very similar to that of the implementation of Prolog by the Warren Abstract Machine [1]. Failure and backtracking occurs when an instruction runs out of answers, or an `Delete_node`[6] or `Assert` instruction fails. The machine then revisits the previous choice made and resumes execution with the next option for the most recent choice.

In general, individual graph transformation rules are implemented by the following pattern of YAM instructions:

(i) Find some nodes or edges from the left-hand side of the rule.

(ii) Check they fulfill any conditions.

(iii) If a full left-hand side has not yet been found then goto (i).

(iv) Execute the changes specified by the rule.

`Assert` instructions in the condition checks make sure that only nodes and edges are selected that match the rule. Any other selection will trigger backtracking.

[6] Nodes can only be deleted if there are no edges incident to them.

4 Implementing the YAM

In our implementation of the YAM, we only store the current graph. It is the purpose of the graph change stack to recover older versions of the graph. Each frame on the graph change stack describes one change that has been made to the graph in such a way that it is possible to undo that change. When backtracking occurs, the choice stack and graph change stack can be unwound in parallel, recovering older graphs as older choices are revisited. It would be infeasible to store entire graphs for choice points, given the number of choices involved in a typical program.

The YAM calls for a graph data-structure with very quick query operations because these will typically be used many more times than update operations. Our implementation achieves this by a quite complex representation of graphs, involving node and edge structures, hashtables and ordered lists.

An edge structure consists of an identifier (a unique integer), the label of the edge, and the identifiers of the start and end node of the edge.

A node structure contains an identifier and the node's label, the indegree and outdegree of the node, and four hashtables, viz. the inedges and outedges indexed by node label and edge label. [7] In these hashtables, the keys are labels and the values are ordered lists of integers (edge identifiers). Because they are ordered, taking the intersection of two lists is a very quick operation. (For example, taking the intersection of those outedges where the target node is labelled "n" and the edge is labelled "e".)

The graph data-structure has two integer-valued functions, *next unused node identifier* and *next unused edge identifier*, which provide fresh identifiers. The structure also has four hashtables. Two tables are the actual stores for nodes and edges, they have identifiers as keys and node or edge pointers as values. The other two tables are mappings from labels to ordered lists of identifiers for nodes and edges (similar to those in the nodes).

Using this structure, graph updates are quite slow—the slowest operation is relabelling a node, which requires time proportional to the size of the neighbourhood of the node being relabelled. But instructions which query the structure of the graph are very quick, and instructions with multiple answers return their results in a fixed order for a given graph. It is possible to compute the i^{th} result of the instruction "A node with label n", or "An edge from node 1 with label e" quickly: the first example requires one hash lookup, and then i steps down the ordered list; the second requires two hash lookups (one to find the node pointer, and one to find the ordered list of edges) and then i steps down the ordered list.

5 Stacks

All components of a YAM state other than the current graph, the program counter and the try and step numbers are maintained in stacks. There are three distinct

[7] That is, inedges by node label, inedges by edge label, outedges by node label and outedges by edge label.

stacks, although the data stack gets copied and saved as part of the backtracking algorithm.

5.1 Data Stack

In a manner very similar to that of the Forth language [2], the abstract machine utilises a data stack for the storage and later retrieval of simple data (integers and strings). Nearly all of the instructions operate on the data stack in some way. There are simple integer operations, such as `Add` and `Divide` (which work on the top two items of the data stack), stack manipulation operations such as `Pick` and `Drop` (which rotate, copy or destroy parts of the stack), and the graph query operations `N?` and `E???`[8] (which search the graph for particular structures). All of these operations have some effect on the data stack, they usually pop their arguments and push their results. In operation the data stack does not get very big and it is normally possible to determine its maximal size statically. This is because the only instructions with a variable effect on the stack are `Pick`, `Roll` and `UnRoll` which pop some integer i and then copy up the ith item of the stack, rotate the top i items "forwards", or rotate the top i items "backwards", respectively. These three instructions are almost always preceded with an integer (`PushI`) which controls their behaviour.

5.2 Choice Stack

Backtracking is implemented using a choice stack. Whenever an instruction returns only one of a number of possible results (such as in the `NA` "Any Node" instruction), a frame is pushed onto the choice stack describing the current state of the system. The frame contains a copy of the current data stack, the program counter, the step number and the try number. Later, if the choice needs reconsidering, the state of the machine can be restored and the next choice tried.

5.3 Graph Change Stack

The state stored in a frame on the choice stack does not save the current state of the graph. To copy and store the entire graph whenever a choice is made or reconsidered would quickly exceed the memory available to any implementation. For this reason there is the graph change stack. Whenever a graph-modifying instruction is executed, a new frame is pushed onto the graph change stack. This new frame describes exactly how to undo the changes made to the graph. When backtracking occurs, in addition to restoring program counter, data stack and try number, frames are popped off the graph stack (and the appropriate changes made to the working graph) until the step number of the top of the graph change stack is smaller than the step number of the state being restored. Hence the graph will have been restored correctly at this point in the execution history.

[8] In these instructions, `?` is a wildcard for `L`, `A`, or `I`. For example the instruction `EILA` is short for "An Edge from Node with Identifier i to Label j via Any edge."

6 Related work

This section briefly discusses the relation of the YAM to graph transformation languages other than GP.

AGG [3] is a Java-based system for graph transformation. A distinctive feature of AGG are rules with negative application conditions, which specify a graph that must not be in the current graph in order for the rule to match. Such a forbidden subgraph can be expressed in YAM code. More challenging are AGG's attributes imported from Java which are not present in the YAM language (as they are incompatible with our leitmotiv of semantic simplicity).

A similar remark applies to the attributes imported from C in PROGRES [7]—a complex graph transformation languages with some involved features. Whilst some of PROGRES' constructs will be easily expressible in YAM code, many will not. For example, the language's type system with graph schemata and path expressions has currently no counterpart in the YAM. According to [7,8], PROGRES is compiled to bytecode of an abstract machine—but we are not aware of a description of this machine.

The FUJABA language [4] is grown from PROGRES but abandons backtracking because "extensive experiences have shown that it is seldom used". Backtracking is needed in the YAM, however, to implement GP's nondeterministic semantics (given in [6]).

7 Conclusion and future work

The implementation of GP [6] will be based on the YAM. A compiler for converting GP programs into YAM code is under development. Experiments show that the machine executes programs much quicker than earlier GP implementation attempts, which took an interpreter approach. This is because most of the rule analysis (such as determining which nodes and edges in a rule get added, deleted or renamed) is done at compile time whereas at run time, the YAM doesn't need to do any analysis.

The current implementation of the YAM is written in about 3000 lines of C code. C was chosen because it is relatively low-level and efficient, and allows to control memory management completely. As an indication of execution behaviour, running the `VertexColouring` program on a 100 node, 300 edge random graph[9] takes approximately 145000 single instruction executions (including backtracking) and 464 graph changes (all of them node renamings), and involves 1658 choice points. On a 2.4 GHz PC with 512 Mb of memory this execution takes less than 0.1 seconds. Running `VertexColouring` on a 1000 node, 3000 edge random graph requires approximately 10.7 million steps and 4350 graph changes, involves 15402 choice points, and takes about 11 seconds.

The YAM makes a useful abstraction. For example, the machine was not designed with GP's while-loop in mind. However the machine needs no modification to accomodate it, as the behaviour of the while construct can be captured at compile

[9] Obtained by creating 100 nodes and adding an edge between two random nodes 300 times.

time. We expect this to be the case for some other constructs that will be added to GP, too.

Future versions of the YAM will provide support for recursive procedures and a type system for graphs because these features will be incorporated in GP. It will also be useful to equip the YAM with a user-friendly interface and to couple the machine with an animation component showing one or all possible executions of a given program.

There is also scope for static analysis of the YAM code: each decision point can be found, and the number of choices to make at any given point can be related to the size of the current graph. Hence, if the number of iterations of rules and while-loops can be estimated or calculated, then it is possible to give time bounds for programs in terms of the size of the input graph.

In the spirit of GP having a simple semantics, it is also the topic of future work to produce a simple, formal semantics for all elements of the YAM.

References

[1] Hassan Aït-Kaci. *Warren's abstract machine: a tutorial reconstruction*. MIT Press, 1991.

[2] Leo Brodie. *Starting Forth: an introduction to the Forth language and operating system for beginners and professionals*. Prentice Hall, 1981.

[3] Claudia Ermel, Michael Rudolf, and Gabi Taentzer. The AGG approach: Language and environment. In H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors, *Handbook of Graph Grammars and Computing by Graph Transformation*, volume 2, chapter 14, pages 551–603. World Scientific, 1999.

[4] Thorsten Fischer, Jörg Niere, Lars Torunski, and Albert Zündorf. Story Diagrams: A new graph rewrite language based on the Unified Modeling Language. In *Theory and Application of Graph Transformations (TAGT'98), Selected Papers*, volume 1764 of *Lecture Notes in Computer Science*, pages 296–309. Springer-Verlag, 2000.

[5] Annegret Habel and Detlef Plump. Computational completeness of programming languages based on graph transformation. In *Proc. Foundations of Software Science and Computation Structures (FOSSACS 2001)*, volume 2030 of *Lecture Notes in Computer Science*, pages 230–245. Springer-Verlag, 2001.

[6] Detlef Plump and Sandra Steinert. Towards graph programs for graph algorithms. In *Proc. International Conference on Graph Transformation (ICGT 2004)*, volume 3256 of *Lecture Notes in Computer Science*, pages 128–143. Springer-Verlag, 2004.

[7] Andy Schürr, Andreas Winter, and Albert Zündorf. The PROGRES approach: Language and environment. In H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors, *Handbook of Graph Grammars and Computing by Graph Transformation*, volume 2, chapter 13, pages 487–550. World Scientific, 1999.

[8] Albert Zündorf. *Programmierte Graphersetzungssysteme*. Doctoral dissertation, RWTH Aachen, Deutscher Universitäts-Verlag, 1996. In German.

Appendix: Hand-compiled code for `VertexColouring`

```
//a macro to find an edge with start and end
//node labels identical.
:EXXA NA Dup Get_node_label Swap EILA ;
Init!: InitEnd ALAP
NA
Dup Get_node_label ATOI 1 Equals Not Assert
```

```
1 ITOA Relabel_node
Init! Jump
InitEnd: Init! Cut
Inc!: End ALAP
Inc1: Inc2 Choice
EXXA
Dup Is_loop Not Assert
Get_edge_end Dup Get_node_label ATOI 1 Add ITOA Relabel_node
Inc! Jump
Inc2:
EXXA
Dup Is_loop Not Assert
Get_edge_start Dup Get_node_label ATOI 1 Add ITOA Relabel_node
Inc! Jump
End:
NoOperation
```