Solving the Compiler Optimisation Case with GROOVE

Arend Rensink Eduardo Zambon

Department of Computer Science
University of Twente, The Netherlands
{rensink, zambon}@cs.utwente.nl

1 Introduction

This report presents a partial solution to the Compiler Optimisation case study using GROOVE. We explain how the input graphs provided with the case study were adapted into a GROOVE representation and we describe an initial solution for Task 1. This solution allows us to automatically reproduce the steps of the constant folding example given in the case description. We did not solve Task 2.

2 GROOVE

GROOVE[1] [1] is a general purpose graph transformation tool set that uses simple labelled graphs. The core functionality of GROOVE is to recursively apply all rules from a predefined set (the graph production system – GPS) to a given start graph, and to all graphs generated by such applications. This results in a *state space* consisting of the generated graphs.

The main component of the GROOVE tool set is the Simulator, a graphical tool that integrates (among others) the functionalities of rule and host graph editing, and of interactive or automatic state space exploration.

2.1 Host Graphs

In GROOVE, the host graphs, i.e., the graphs to be transformed, are simple graphs with nodes and directed labelled edges. In simple graphs, edges do not have an identity, and therefore parallel edges (i.e., edges with same label, and source and target nodes) are not allowed. Also, for the same reason, edges may not have attributes.

In the graphical representation, nodes are depicted as rectangles and edges as binary arrows between two nodes. Node labels can be either node types or flags. Node types [resp. flags] are displayed in **bold** [resp. *italic*] inside a node rectangle.

2.2 Rules

The transformation rules in GROOVE are represented by a single graph and colours and shapes are used to distinguish different elements. Figure 1 shows a small example rule.

- **Readers.** The black (continuous thin) nodes and edges must be present in the host graph for the rule to be applicable and are preserved by the rule application;

[1] Available at http://groove.cs.utwente.nl

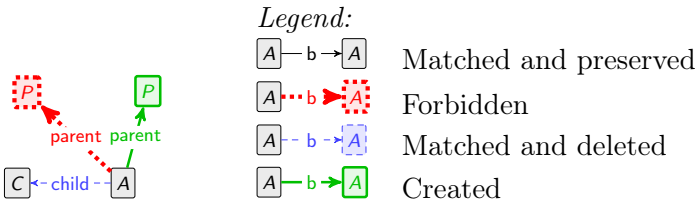

Figure 1: Example GROOVE rule and legend

- **Embargoes.** The red (dashed fat) nodes and edges must be absent in the host graph for the rule to be applicable;
- **Erasers.** The blue (dashed thin) nodes and edges must be present in the host graph for the rule to be applicable and are deleted by the rule application;
- **Creators.** The green (continuous fat) nodes and edges are created by the rule application.

Embargo elements are usually called Negative Application Conditions (NACs). When a node type or flag is used in a non-reader element but the node itself is not modified, the node type or flag is prefixed with a character to indicate its role. The characters used are +, −, and !, for creator, eraser, and embargo elements, respectively.

Additional notation and functionalities of the tool are presented along with the developed solution for the case.

3 Solution

In this section we describe a partial solution for Task 1 of the case study.

3.1 Input from the FIRM representation

The input graphs provided with the case study are stored in GXL format and conform to the FIRM representation. GROOVE also uses GXL to store graphs but it was not possible to immediately load the given files because the input graphs have certain properties that are not compatible with GROOVE (e.g., edges with attributes) and therefore require some adaptation.

The case description lists all node and edge types that may occur in the program graphs. These types are also included in the GXL files given on the form of a type graph. Based on these two sources of information we constructed our own type graph[2] in GROOVE, shown in Figure 2.

Each node in the figure corresponds to a node type; some have associated attributes. Types shown in ***bold italic*** inside dashed nodes are abstract. Edges with open triangular arrows indicate type inheritance. A key point in the type graph shown in Figure 2 is the following. In GROOVE edges do not have types or attributes while in FIRM the edges do. To encode these extra properties in GROOVE, edges have to be *nodified*, i.e., each edge of the FIRM graph is transformed to a node in the GROOVE graph with a proper sub-type of ***Edge*** and associated position attribute. Nodes representing operations, i.e., sub-types of ***Node***, are connected via ***Edge*** nodes and associated edges labelled in and out. The remaining elements of the type graph of Figure 2 correspond directly to the ones described in the case study.

After creating our type graph, the program graph used in the constant folding example was manually created by inspecting the given GXL file and the corresponding figure in the case

[2] GROOVE enforces static typing, so there is no overhead for type checking while performing a transformation. Using a type graph is a very convenient way to avoid simple mistakes (e.g., typos) while creating a grammar.

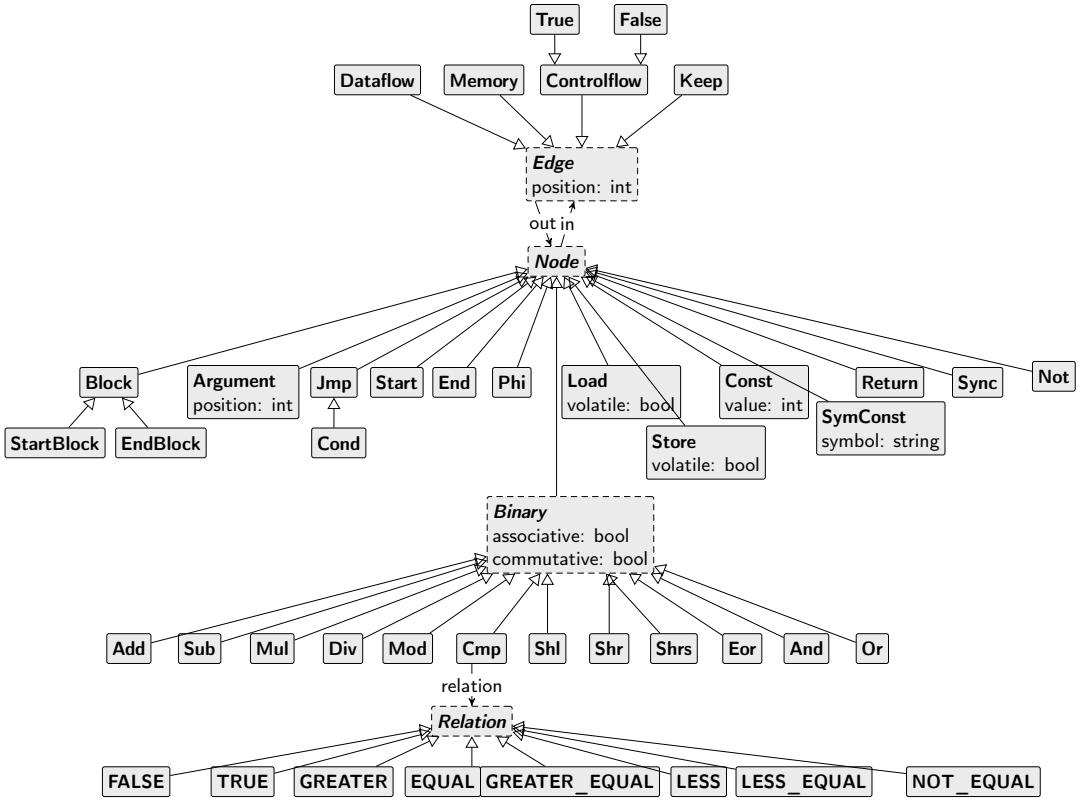

Figure 2: Type graph for the adapted input graphs.

study. Our start graph in a plain representation, i.e., without block containment visualisation, is shown in Figure 3.

The structure of the program graph shown in Figure 3 directly corresponds to the one given in the case description. Edge nodification clutters the representation but, if one so desires, this issue could be handled on the GUI level, with a dedicated display format, in the same vein as is done in the case description.

We would like to point that, despite the current manual adaptation of the input graphs, there are no technical limitations that prevent the automatic loading of Firm graphs using the conversion described above. Automatic loading was not done due to time limitations only.

3.2 Verifier

We implemented the sanity checks described in the case study in negated form such that if an invalid configuration is produced, then a checking rule matches. Figure 4 shows rule consts, that is triggered if there is a constant located in a Block that is not the StartBlock. The other checks given in the case study were implemented with the rules named single-start, single-end, containment, phi-check, and pos-check. (See the grammar for the solution in the SHARE image.)

3.3 Constant Folding

To solve the constant folding example given in the case description we created seven rules to perform the folding of operations and another three cleanup rules to handle dangling edges and constants without references.

Figure 5 shows rule add-fold-int, that performs the last step of the transformation: folding

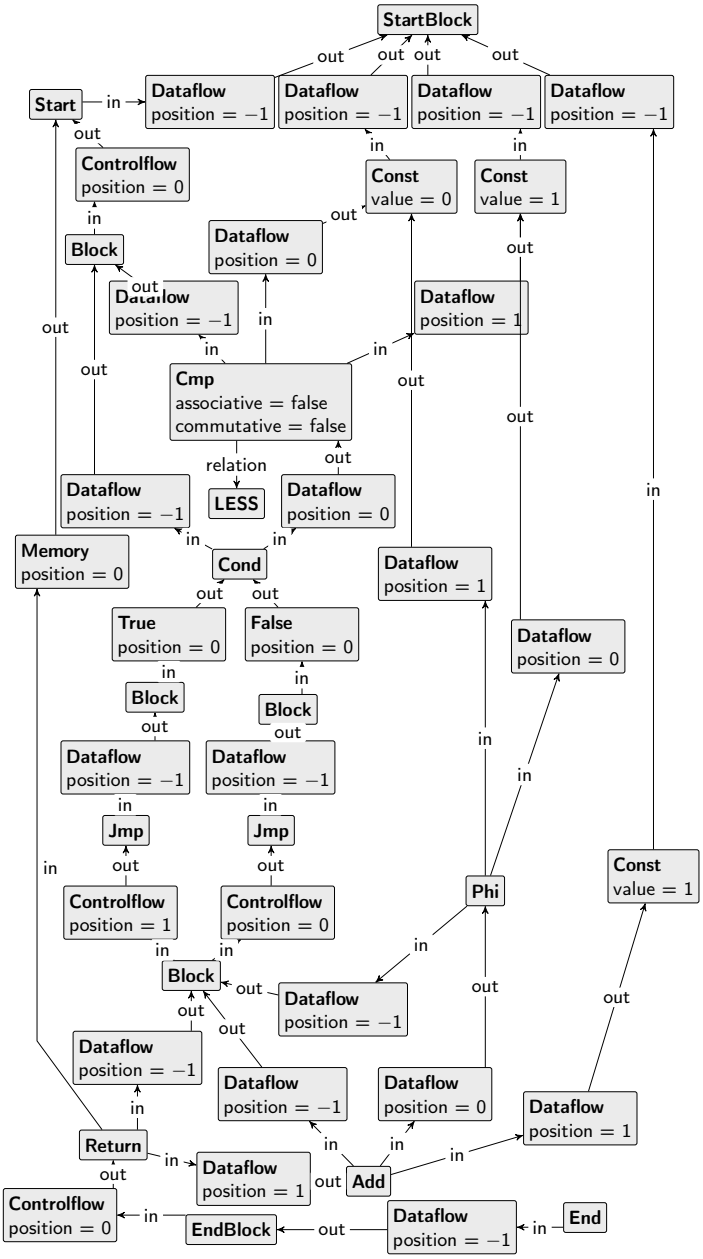

Figure 3: Program graph of minimum plus one function with constants.

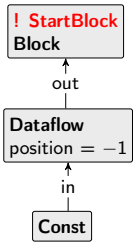

Figure 4: Sanity check rule consts.

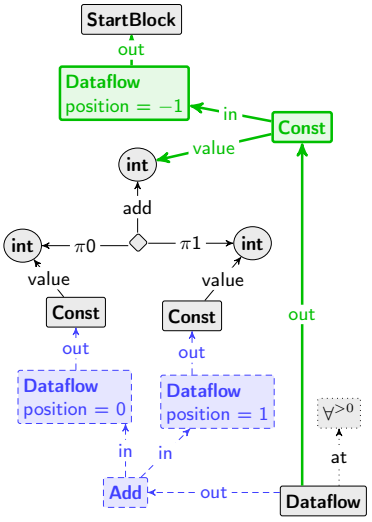

Figure 5: Rule add-fold-int, for folding the addition of two integer constants.

of an Add operation with two constant operands. The Add node and the two Dataflow edges associated with the operands are deleted by the rule. The constants used in the addition are not removed because they may be referenced by other operations. The addition of the two values is performed by the product node (the node with a diamond shape). The two operands are indicated by the edges labelled with π and the result is the value node pointed by the edge labelled add. A new constant is created with the result of the addition and all Dataflow edges incoming into Add are re-routed to the newly created constant. We use a special quantifier node (labelled with $\forall^{>0}$) to redirect an arbitrary number of Dataflow edges.

The remaining rules are similar. We created one rule for each operation folding, except for the handling of unreachable blocks, which uses two rules, one for removing blocks and another for adjusting the edges of Phi nodes. (Again, for the complete solution, we refer the reader to the grammar that is available in the SHARE image.) The final program graph for the running example is shown in Figure 6.

We did not create folding rules for the operations that do not occur in the given example. Still, once more, we do not foresee any technical difficulties to do so. The remaining operations were not handled only due to limited time availability.

4 Conclusion

In this report we presented the key points of our solution for Task 1 of the case study. Task 2 was not addressed. We conclude with an overview according to the criteria listed in the case study and we give some constructive criticism for the authors about the case description.

- **Completeness.** The test suite for the case study was not yet available at the time of this writing. Still, since we did not cover all operations, it is expected that no program graph other than the example discussed can be handled. Absence of automatic loading of graphs is another limitation that will prevent the use of the test suite.
- **Performance.** N/A. See reasons in the item above.
- **Conciseness.** As a rule of thumb, we have one rule for each operation.
- **Purity.** The solution is entirely made of graph transformations, no glue code is necessary.

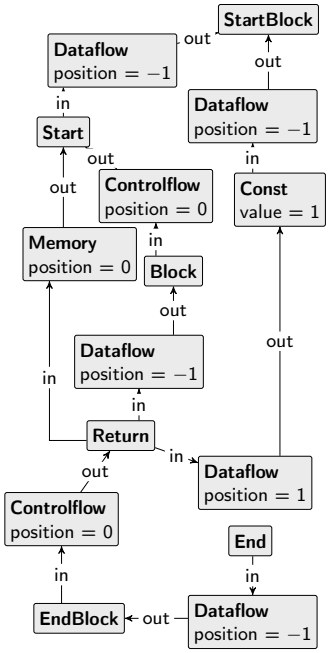

Figure 6: Final configuration for the constant folding example.

4.1 Comments on the case study description

While we understand that creating a case study is a challenging enterprise we still would like to indicate some points of improvement to the authors.

1. **GXL integration.** The given GXL files do not pass XML validation tools. See, e.g., `http://www.xmlvalidation.com/`.

2. **Number of operations.** We agree that the main goal of the case, namely, the performance comparison between tools is interesting. However, the case study itself makes reaching a point where this comparison is possible quite difficult. In particular, are all the operations listed in the case study really necessary? There are almost 30 of them. It seems that there are many details that could be abstracted away, e.g., volatile loads and stores. In addition, there are many binary operations that are virtually the same. If a tool can handle integer addition, it is expected that it will also be able to perform other arithmetical operations in the same way. We understand that due to the nature of this case study these simplifications may be hard to do, but the bottom line is that we believe the participants would like to use the case study to evaluate and improve their tool (for example, by trying to improve its performance) preferably without having to spend many hours solving a problem that is full of details of the original domain.

3. **Task 2.** The description of Task 2 is too concise and therefore quite hard to understand. Maybe for those with a background in compiler construction the task is obvious but for others we believe this is not the case.

References

[1] Ghamarian, A., de Mol, M., Rensink, A., Zambon, E., and Zimakova, M.; *Modelling and analysis using* GROOVE . International Journal on Software Tools for Technology Transfer (STTT). Springer – Berlin, March 2011, `http://dx.doi.org/10.1007/s10009-011-0186-x`.