Journal of Logical and Algebraic Methods in Programming 83 (2014) 225–234

Contents lists available at ScienceDirect

Journal of Logical and Algebraic Methods in Programming

www.elsevier.com/locate/jlamp

Towards “mouldable code” via nested code graph transformation

Wolfram Kahl [1]

McMaster University, Hamilton, Ontario, L8S 4K1, Canada

ARTICLE INFO

Article history:
Available online 12 February 2014

ABSTRACT

Program transformation is currently de facto restricted to abstract syntax tree rewriting. However, many program transformation patterns, in particular in the realm of high-performance code generation, can more naturally be understood and expressed as *graph* transformations. We describe the conceptual organisation of a system based on application of algebraic graph transformation rules to data-flow and control-flow graphs, and outline the work, both theoretical and of implementation nature, that still needs to be done to realise this long-term project.

1. Introduction

An important part of the supporting infrastructure for optimising code generators, in particular in compilers, consists of analyses of graphs, especially control-flow graphs and data-flow graphs. In addition, many of the optimisations enabled by these analyses are then usefully understood as graph transformations.

Interestingly, the program transformation literature almost exclusively concentrates on transformation of (higher-order) abstract syntax *trees*. It is of course customary in a compiler context to parse the given programs into abstract syntax trees, and then extract from these the necessary information to construct the graphs to be used for data-flow and control-flow analyses using attribute grammars, while still considering the abstract syntax trees as the internal representation of the program.

However, many of the transformations that are used for code optimisation, in particular in the back-ends of compilers, act on patterns that can usefully be thought of as *graph* patterns, and the resulting transformations are frequently explained as *graph transformations* applied to the control-flow graph and the data-flow graph, which are usually considered as only *derived* from the programs being transformed.

In this paper, we instead adopt the approach of *representing* (parts of) programs as graphs, in particular as nested graphs with control-flow graphs and data-flow graphs at different levels. Mathematically, we represent all these graphs as directed hypergraphs, and use consistent drawing conventions. In directed hypergraphs, hyperedges (drawn as rectangles) can have multiple input nodes (connected by “input tentacles” drawn as arrows from the nodes to the edges) and multiple output nodes (connected by “output tentacles” drawn as arrows from the edges to the nodes). The graph itself may also have an input/output interface consisting of two lists of nodes; input nodes are indicated by numbered triangles above, and output nodes by numbered triangles below. Fig. 1 shows a data-flow graph (which is therefore acyclic) representing the expression $2 + (7 + (3 + 5 \cdot x) \cdot x) \cdot x$ with x standing for the single input; this evaluates a polynomial at x using Horner’s rule. Fig. 2 is

E-mail address: kahl@cas.mcmaster.ca.

[1] This research is supported by the National Science and Engineering Research Council of Canada, NSERC.

http://dx.doi.org/10.1016/j.jlap.2014.02.010
2352-2208/© 2014 Elsevier Inc. All rights reserved.

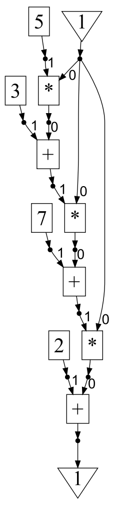

Fig. 1. Data-flow graph.

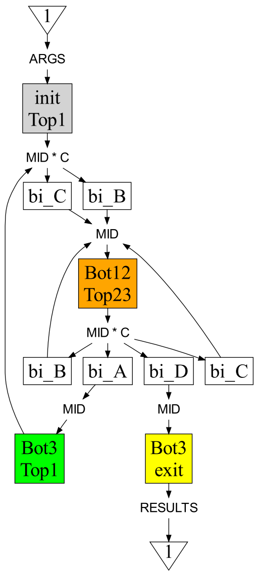

Fig. 2. Control-flow graph.

a control flow graph for a particular way to re-arrange the nested loops of matrix multiplication [1]. Control-flow graphs, like finite automata, have a single start state (drawn as an input), and all hyperedges are actually single-input single-output edges.

1.1. *Data-flow-graph transformations*

For example, common subexpression elimination can be understood as forming the quotient of a data-flow graph by a congruence relation; "dead-variable elimination" can be understood as transitioning to the sub-graph of a data-flow graph that is reachable from the declared output nodes.

Many cases of "strength reduction", i.e., of replacing special cases of more expensive operations with less expensive operations, are graph rewriting rules that correspond directly to term rewriting rules, such as $2*x \to \mathsf{shiftLeft}(x, 2)$. However, non-linear rules such as $x + x \to \mathsf{shiftLeft}(x, 2)$ are more typically thought of as local graph rewriting rules where the two arguments of "+" have to be the same node, than as non-linear term or graph rewriting rules where the two sub-terms matched to x are compared for equality or isomorphism (or, more precisely, bisimilarity).

A more intricate situation arises in SIMD-isation: Assume that SIMD arithmetic for vectors of four elements are available, but a given program contains only three independent additions that can be grouped into a vector operation at a certain point. Since that vector operation will always perform four additions, that forth addition, if considered as an individual operation, is technically dead code since its result will not be used. This particular SIMD-isation transformation can therefore be considered as dead-variable introduction, followed by factoring multiple occurrences of isomorphic subgraphs into a single copy using SIMD operations instead of individual operations, again as data-flow graph transformation.

1.2. *Control-flow-graph transformations*

Many other common optimisation techniques can usefully be understood as control-flow graph transformations, for example:

- Dead-code elimination can be understood as transitioning to the sub-graph of a control-flow graph that is reachable from the declared entry nodes.
- Loop unrolling can be used to reduce the run-time impact of loop overhead; it will typically be a control-flow transformation that also affects the loop control logic, which may therefore also affect some data manipulations.

One of the purposes of many control-flow graph transformations, including loop unrolling, is to enable further data-flow optimisations. For this purpose, it makes sense to consider the edges of the control-flow graph to be labelled with data-flow graphs corresponding to "straight-line code", also called "basic blocks".[2]

[2] Integrating both data-flow and control-flow aspects in a single non-nested graph structure is feasible only for very restricted purposes. In general, one particularly obvious problem is that data-flow "sharing" and control-flow "branching" are incompatible interpretations of multiple outgoing edges.