Instruction Selection by Graph Transformation

Sebastian Buchwald
Karlsruhe Institute of Technology (KIT)
76128 Karlsruhe, Germany
buchwald@kit.edu

Andreas Zwinkau
Karlsruhe Institute of Technology (KIT)
76128 Karlsruhe, Germany
zwinkau@kit.edu

ABSTRACT

Common generated instruction selections are based on tree pattern matching, but modern and custom architectures feature instructions, which cannot be covered by trees. To overcome this limitation, we are the first to employ graph transformation, the natural generalization of tree rewriting. Currently, the only approach allowing us to pair graph-based instruction selection with linear time complexity is the mapping to the Partitioned Boolean Quadratic Problem (PBQP). We present formal foundations to verify this approach and therewith identify two problems of the common method and resolve them. We confirm the capabilities of PBQP-based instruction selection by a comparison with a finely-tuned hand-written instruction selection.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*Correctness proofs*; D.3.4 [**Programming Languages**]: Processors—*Compilers,Retargetable Compilers*

General Terms

Algorithms, Theory, Verification

1. INTRODUCTION

Adapting a compiler to new hardware architectures is a common task, especially in embedded systems. Generating a compiler from an architecture description is an appealing improvement to manual programming. Yet popular compilers do not employ this technique as it results in poorer code quality. New approaches like those based on solving the Partitioned Boolean Quadratic Problem (PBQP) suggest that this trade-off may not be necessary [7, 8, 19].

The most architecture-specific part of code generation is instruction selection, where the intermediate representation (IR) gets transformed to architecture-specific instructions. For every machine instruction there is a corresponding IR pattern with equal semantics. A cost model weighs the patterns against each other, considering e.g. code size, execution time or energy use. In the IR the patterns have to be identified and a subset with minimal cost has to be selected that covers the whole program.

Modern compilers use an IR in static single assignment (SSA) form [6, 27] allowing a graph-based representation called *program graph* or "sea of nodes" [4]. Thus, instruction selection can be considered as graph transformation.

Contrary to classical algorithms for generated instruction selection, PBQP can model program and pattern graphs. With this increased flexibility instruction selection can support more complex operations (vector operations, custom instructions, etc.), which cannot be processed by tree rewriting, to achieve higher code quality. A PBQP-based instruction selection can therefore combine the code quality of manually fine-tuned compilers with the short compiler development times desired in embedded systems.

Least-cost instruction selection on directed acyclic program graphs is NP-complete [26]. Therefore, a graph-based linear-time instruction selector must sometimes make heuristic decisions to be efficient. The PBQP is a modelling of the instruction selection problem that reduces the problem to its core before heuristic decisions are made.

However, finding *any* PBQP solution is NP-complete [19] in general. This seems to contradict the empirical observation that a linear algorithm always finds a solution. So the question arises, whether PBQP instances, which are constructed during instruction selection, form a special subclass of the PBQP that can be solved in linear time. Previous work ignored this question, but not finding a solution is no alternative, because a fallback instruction selection would be required, which invalidates the argument of reduced development time. Our contributions are as follows:

- We provide a new formal representation for instruction selection based on graph transformation.
- We show that the PBQP-based approach can indeed fail and provide a rectification for the PBQP solving algorithm. Additionally, we identify sufficient preconditions for the rule sets of architecture descriptions to guarantee that a solution is found.
- We derive a method for automatic completion of insufficient rule sets, which simplifies the specification of architecture descriptions.
- An evaluation of PBQP-based instruction selection on the IA32 architecture confirms the competitive performance of this approach.

CASES'10, October 24–29, 2010, Scottsdale, Arizona, USA.

The paper is organized as follows: In Section 2, we describe our formal model of graph-based instruction selection. Section 3 explains the PBQP and how it is used for instruction selection. Section 4 depicts two problems of the known approach and shows how to solve them. Our prototype is evaluated in Section 5. We summarize related and future work in Section 6 and Section 7, respectively. Finally, we conclude in Section 8.

2. GRAPH-BASED INSTRUCTION SELECTION

Instruction selection can be divided into three phases:

Pattern matching: Find all occurrences of the given pattern set in the current program graph.

Selection: Select an appropriate subset of the matches that covers the whole program graph.

Replace: Replace the selected patterns with their corresponding instruction.

PBQP solving can only account for the selection phase, which means the pattern matching and replace phases are not considered so far. Since PBQP-based selection can even handle DAG-shaped patterns, we replace the usual tree rewriting system [14, 23, 24] with a more general graph rewriting system [11, 18, 28]. More precisely, we use the algebraic single-pushout approach [22], which is based on category theory [12].

2.1 Formal graph-foundations

A category consists of objects and morphisms. We embed program graphs and pattern graphs into a joint category with IR graphs as objects.

DEFINITION 1 (IR GRAPH). *Let Σ be a finite set of node types. An IR graph G is a 6-tuple $(V_G, E_G, \mathrm{src}, \mathrm{tgt}, \mathrm{pos}, \mathrm{type})$, where*

- *V_G is a finite set of nodes*
- *E_G is a finite set of edges*
- *$\mathrm{src} : E_G \to V_G$ assigns each edge its source node*
- *$\mathrm{tgt} : E_G \to V_G$ assigns each edge its target node*
- *$\mathrm{pos} : E_G \to \mathbb{N}_0$ assigns each edge its position*
- *$\mathrm{type} : V_G \to \Sigma$ is a partial typing of nodes*

For an untyped node v we denote $\mathrm{type}(v) = \bot$. Every type $t \in \Sigma$ has a static out-degree $\deg(t) \in \mathbb{N}_0$. Likewise, we assign each node v an out-degree $\deg(v) = |\{e \in E_G \mid \mathrm{src}(e) = v\}|$, which equals the out-degree of the assigned type:

$$\mathrm{type}(v) \neq \bot \Rightarrow \deg(v) = \deg(\mathrm{type}(v)).$$

Furthermore, we impose two additional conditions for the position of an edge $e \in E_G$:

$$0 \leq \mathrm{pos}(e) < \deg(\mathrm{src}(e))$$
$$\forall e' \in E_G : \mathrm{src}(e) = \mathrm{src}(e') \wedge \mathrm{pos}(e) = \mathrm{pos}(e') \Rightarrow e = e'.$$

DEFINITION 2 (MORPHISMS ON IR GRAPHS). *A morphism $f : G \to G'$ is a pair (f_V, f_E) of mappings $f_V : V_G \to V_{G'}$ and $f_E : E_G \to E_{G'}$, where*

$$\begin{aligned} f_V(\mathrm{src}(e)) &= \mathrm{src}(f_E(e)) \\ f_V(\mathrm{tgt}(e)) &= \mathrm{tgt}(f_E(e)) \\ \mathrm{pos}(e) &= \mathrm{pos}(f_E(e)) \\ \mathrm{type}(v) \neq \bot &\Rightarrow \mathrm{type}(v) = \mathrm{type}(f_V(v)). \end{aligned}$$

DEFINITION 3 (ISOMORPHIC IR GRAPHS). *IR graphs G and G' are called isomorphic $G \cong G'$, if there are two morphism $f : G \to G'$ and $g : G' \to G$ with*

$$f \circ g = \mathrm{id}_{G'}$$
$$g \circ f = \mathrm{id}_G.$$

DEFINITION 4 (PROGRAM GRAPH). *A program graph G is an acyclic IR graph with $\mathrm{type}(v) \neq \bot$ for each node $v \in V_G$.*

Usual program representations are cyclic due to control flow cycles and only acyclic within a basic block. If the IR is in SSA form, the restriction to acyclic graphs in Definition 4 still allows instruction selection for whole functions, because each loop contains at least one ϕ-node, which we assume is not modified by instruction selection. Thus, for formal[1] considerations we split a ϕ-node into two nodes. One node has no incoming edges and the k outgoing edges of the original node, so let us call it ϕ_k. The other node has all the incoming edges of the original node and no outgoing edges, effectively it is a ϕ_0-node. To keep the number of the resulting ϕ_k-types finite we employ the limitations provided by the language specifications.

DEFINITION 5 (PATTERN GRAPH). *A pattern graph P is a rooted, acyclic IR graph with root $\mathrm{rt}(P)$, where*

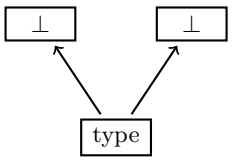
$$\mathrm{type}(\mathrm{rt}(P)) \neq \bot$$
$$\mathrm{type}(v) = \bot \Rightarrow \deg(v) = 0$$

for all nodes $v \in V_P$.

In contrast to program graphs, the leaf nodes of pattern graphs can be untyped. According to Definition 2, untyped nodes can be mapped to nodes of arbitrary type. Intuitively, an untyped node represents a value stored in a register. The mapping of a pattern into the program graph is called a match. To find such matches is called pattern matching and describes the first phase of instruction selection.

DEFINITION 6 (MATCH). *A match (P, ι) of an IR graph G consists of a pattern graph P and an injective morphism $\iota : P \hookrightarrow G$.*

The simplest kind of a pattern graph consists of a typed root with only untyped operands, if any.



DEFINITION 7 (ATOMIC PATTERN GRAPH). *A pattern graph P is called atomic, if $\mathrm{type}(v) = \bot$ holds for all nodes $v \in V_P \setminus \{\mathrm{rt}(P)\}$.*

[1] In praxis, the split has the same effect as allowing exactly one alternative at a ϕ-node: an atomic ϕ-pattern.

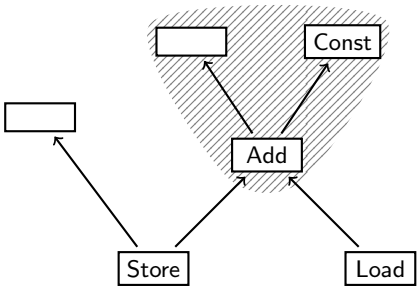


Figure 1: Load and Store with a common address calculation.

2.2 Covering a program graph

For non-atomic and more complex pattern graphs we want to allow multiple pattern graphs to overlap in the program graph. Figure 1 shows an address calculation, which is used by multiple operations. For the Store and Load nodes pattern graphs exist, which overlap on the marked subgraph. Using these patterns avoids to materialize the address in a register and thus can be advantageous regarding code size and register pressure. However, some nodes (volatile Loads for example) must never be in the intersection of overlapping patterns. Thus, the pattern matcher has to reject such matches, if the critical node has multiple predecessors[2].

Definition 8 (Spanning graph). *Let G be an IR graph and $v \in V_G$ a node. Furthermore, let*

$$S(v) = \{v' \in V_G \mid \text{there is a path from } v \text{ to } v'\}$$

be the set of nodes reachable from v. The spanning graph $\text{span}(v)$ *is the induced subgraph of $S(v)$.*

Using the concept of spanning graphs, we can relate nodes of different pattern graphs, to formally grasp overlaps of patterns. Root nodes are explicitly excluded from this relation, apart from the fact that every node is equivalent to itself. The example in Figure 1 marks the spanning graph of the Add node.

Definition 9 (Equivalence of pattern nodes). *Let $(P_i)_{i \in I}$ be a family of pattern graphs with $v_i \in V_{P_i}$ and $v_j \in V_{P_j}$. We define an equivalence relation on $\bigcup_{i \in I} V_{P_i}$ by*

$$v_i \sim v_j \Leftrightarrow v_i = v_j \vee (\text{span}(v_i) \cong \text{span}(v_j) \wedge \text{rt}(P_i) \neq v_i \wedge \text{rt}(P_j) \neq v_j). \quad (1)$$

We describe the relation between multiple patterns and a program graph using morphisms. We define the term cover, which relates the two graph types, first for nodes and then for acyclic IR graphs. Remember that the overlap of patterns can lead to a situation where a node is covered by multiple patterns simultaneously.

Definition 10 (Cover of a node). *Let* $\text{im}(\iota)$ *denote the image of ι and ι^{-1} the inverse morphism of ι. The cover C_v of a node $v \in V_G$ in an acyclic IR graph G by a set of pattern graphs $\mathcal{P}$ is a non-empty set of matches (P_i, ι_i) of G with $P_i \in \mathcal{P}$. For every match there must be a preimage of v with the same type:*

$$\forall i : v \in \text{im}(\iota_i) \wedge \text{type}(\iota_i^{-1}(v)) = \text{type}(v), \quad (2)$$

where $\bot = \bot$ is true. Additionally, for all matches (P_i, ι_i) and (P_j, ι_j) in C_v the preimages of v must be equivalent:

$$\iota_i^{-1}(v) \sim \iota_j^{-1}(v). \quad (3)$$

Figure 2 illustrates a cover where a program graph and two pattern graphs P_A and P_B are shown. According to the definition above, $\{(P_A, \iota_A), (P_B, \iota_B)\}$ is a cover for the Shl and the Const node. These nodes have two preimages, since they are covered by two patterns simultanuously. The Const and the Shl nodes of the patterns are equivalent, respectively, because their spanning graphs in the patterns are isomorphic. In contrast, the Add node in P_A is a root node, therefore by Definition 9 it is not equivalent to the Add node in P_B, which is no root, hence the set of matches violates Equation 3.

Definition 11 (Cover of an IR graph). *A set C_G of matches (P_i, ι_i) is called cover of an acyclic IR graph G, if for each node $v \in V_G$ the set*

$$C_v = \{(P, \iota) \in C_G \mid v \in \text{im}(\iota) \wedge \text{type}(\iota^{-1}(v)) = \text{type}(v)\}$$

is a cover of v. Additionally, every node $v \in V_G$ must satisfy the condition

$$(\text{type}(v) \neq \bot \wedge \ \exists i : v \in \text{im}(\iota_i) \wedge \text{type}(\iota_i^{-1}(v)) = \bot) \Rightarrow \exists j : \iota_j(\text{rt}(P_j)) = v. \quad (4)$$

The condition in Equation 4 means that every typed node v, which is covered by a leaf node without type, must also be covered by the root node of another pattern. Together with the previous definitions, it also implies that if a root node w covers a node v, every other pattern node u, which also covers v, must be untyped.

For an example consider Figure 2 again, where the pattern graphs P_A and P_B cannot both be contained in a cover. Definition 11 requires for $v = \text{Add}$ that $C_v = \{(P_A, \iota_A), (P_B, \iota_B)\}$ is a cover of the Add node, which violates Equation 3 as we already know.

2.3 Rewrite rules

For a complete instruction selection we also need the replace phase. A rewrite rule is a transformation of one pattern graph into another one. The specification of an instruction selection, which is the major part of an architecture description, is a set of rewrite rules transforming nodes with IR types into nodes with types corresponding to operations of the target architecture.

Instruction selection can now be defined solely on the level of graph transformations. Given is a set of rewrite rules, a program graph G and a cost model. The pattern set is extracted from the rule set and a set of all matches $\mathcal{M}$ is found in the program graph. The *instruction selection problem* is to select a cover $C_G \subseteq \mathcal{M}$ of G with minimal cost. To select this cover, we map the problem to the PBQP, as shown in the next section.

3. SELECTION WITH PBQP

PBQP-based instruction selection gets its name by mapping the instruction selection problem to the Partitioned

[2]In graph transformation theory this can be expressed using negative application conditions (NACs) [16].

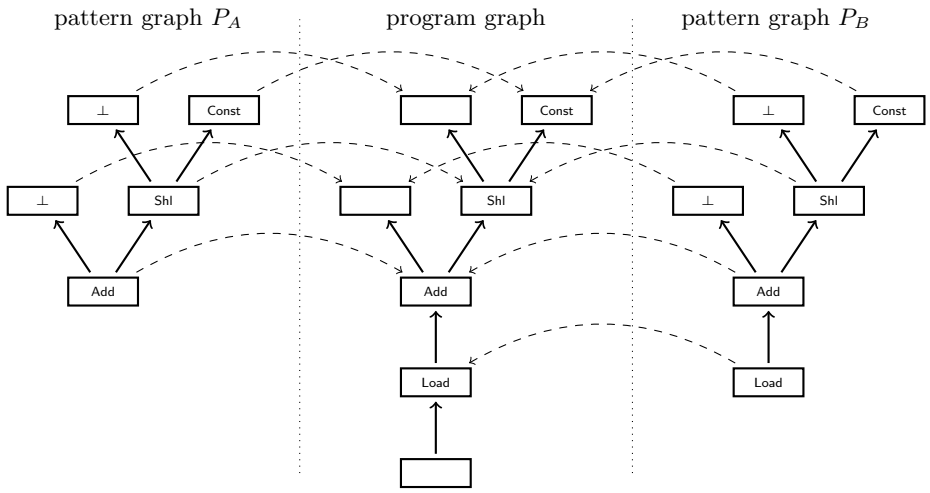


Figure 2: Cover of nodes by patterns.

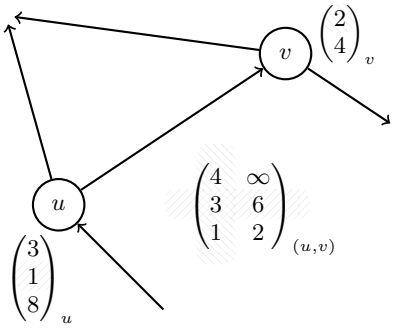


Figure 3: Basic PBQP example.

Boolean Quadratic Problem. In the following, we give an overview of the PBQP itself and the algorithm for solving PBQP instances as known from previous work [8, 9, 19, 29]. Then we show how to construct PBQP instances within the scope of graph-based instruction selection.

3.1 Partitioned Boolean Quadratic Problem

The PBQP is an abstract optimization problem, where one has to make multiple, interdependent choices with minimal cost.

Formally, let $G = (V, E)$ be a directed graph with a totally ordered set of nodes V and a set of edges $E \subseteq \{(u, v) \mid u, v \in V \wedge u < v\}$. For every node $u \in V$ there is a set of alternatives $\mathcal{A}_u = \{A_1, \ldots, A_l\}$, whose costs are specified by the vector $\vec{c}_u \in \mathbb{R}^l$. For every edge (u, v) a cost matrix $C_{(u,v)} \in (\mathbb{R} \cup \{\infty\})^{k \times l}$ is assigned[3], where k and l are the number of alternatives of u and v, respectively. If an alternative A_i at an edge (u, v) of source node u is selected, then the i-th row of the corresponding cost matrix $C_{(u,v)}$ must be selected, too. Likewise, the selection at the target node v determines the column of the cost matrix. Figure 3 shows two nodes, where an alternative (with costs 1 and 2, respectively) in their corresponding cost vectors is selected. With this selection the row and column of the cost matrix is determined to have the value 3 for a total sum of $1+2+3=6$ for this part.

A *selection* in a PBQP instance is a mapping $v \mapsto A_i \in \mathcal{A}_v$ that assigns each node an eligible alternative. The cost of a selection is the sum of all chosen vector and matrix entries. A selection is called a *solution* if its cost is finite. For example, every selection in Figure 3 containing $\{u \mapsto A_1, v \mapsto A_2\}$ is not a solution, because an entry with cost ∞ is selected in the matrix. This means that infinite cost entries can be used to express incompatibility between alternatives. The goal of the PBQP is to find a solution with minimal cost.

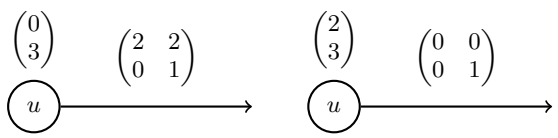
3.2 Solving PBQP instances

The algorithm for solving PBQP can be divided into three steps:

1. Reduce the graph until no edges are left.
2. Locally select a (globally) minimal alternative at every node.
3. Reinsert reduced nodes in reverse order and select a (locally) minimal alternative at every accessory node.

The easiest way of reducing the graph is to delete an edge with a zero cost matrix. In order to obtain such a matrix we shift costs from the matrix into the adjacent vectors. A cost amount of x can be accounted in the entry A_i of a cost vector or in the entries in the i-th row/column of an adjacent cost matrix. In both cases the total cost of a corresponding selection is the same. Figure 4 depicts an example for $x = 2$, where the costs are shifted between the first row of the cost matrix and the first entry of the cost vector. As a result of shifting costs into the vector, the cost matrix is in *normal form*, i.e. the minimum cost entry of each row and column equals zero. If normalisation would produce infinite vector costs, the corresponding alternative (including cost matrix rows/columns) is deleted. The deletion of a matrix row/column may induce further changes in the adjacent vector, hence the normalisation may propagate throughout the whole graph and delete alternatives. Altogether, we obtain the following reduction:

[3]More precisely, a Min-Plus-Algebra [5] has to be used, because of the inclusion of ∞. A more detailed definition [2] is not necessary here, but there are subtle algebraic implications.



(a) Original cost matrix. (b) Normalized cost matrix.

Figure 4: Shifting costs between vector and matrix.

RE: Independent edges have a cost matrix that can be normalized into a zero matrix and can be removed after the normalization is processed.

The basic idea of the following two reductions is the transformation to a smaller PBQP instance with equal minimal costs, such that each solution of the small instance can be extended to a solution of the original instance with equal costs.

R1: Nodes of degree one can be removed, after costs are accounted in the adjacent node.

R2: Nodes of degree two can be removed, after costs are accounted in the cost matrix of the edge between the two neighbors; If necessary, the edge is created first.

For sparse graphs these reductions are very powerful, since they diminish the problem to its core. If the entire graph can be reduced by RE, R1, and R2 [29], then the resulting PBQP solution is optimal. If nodes of degree three or higher remain, a heuristic ensures linear time behavior:

RN: Nodes of degree three or higher. For a node u of maximum degree we select a locally minimal alternative, which means we only consider u and its neighbors. After the alternative is selected, all other alternatives are deleted and the incident edges are independent, so they can be removed by using RE.

Since the heuristic reduction does not take edges between neighbors into account, we want to have as much information as possible in the cost vectors. This can be achieved by normalizing each matrix before a heuristic reduction is applied.

3.3 Construction of the PBQP graph

In this section, we describe the construction of a PBQP instance from a program graph G and a set of pattern graphs $\mathcal{P}$. First a PBQP graph (V, E) is built from the program graph G, where

$$\begin{aligned} V &= V_G \\ E &= \{(\mathrm{src}(e), \mathrm{tgt}(e)) \mid e \in E_G\}. \end{aligned}$$

Since our program graphs are considered acyclic, there is a total order in V, where $\mathrm{src}(e) < \mathrm{tgt}(e)$ for all edges $e \in E_G$ and (V, E) satisfies the conditions for a PBQP graph. While a program graph can have multiple edges with the same source and target node, these edges are merged to reduce the node degree. Nodes are mapped 1:1.

Furthermore, we need to construct a vector of alternatives for every node v, where equivalent matches are combined to one alternative. Two matches are equivalent $\sim_v$ with respect to v if and only if the corresponding preimages of v are equivalent according to Equation 3. Let $\mathcal{C}_v$ denote the set of all matches (P, ι), such that each $\{(P, \iota)\}$ is a cover of v, then the alternatives of v are defined by

$$\mathcal{A}_v = \mathcal{C}_v / \sim_v .$$

Every pattern has finite costs, given by an external cost model or a specification. These costs are put into the root node of the pattern, because it is not equivalent to any other node by Definition 9. All other nodes of the pattern have costs 0.

For the declaration of the cost matrices we consider one element (P_u, ι_u) of an alternative A_u of the source node u and one element (P_v, ι_v) of the alternative A_v of the target node v independently of other combinations. The corresponding matrix entry is constructed as follows:

$$c(A_u, A_v) = \begin{cases} \infty & \mathrm{type}(\iota_u^{-1}(v)) = \bot \wedge\ \iota_v^{-1}(v) \neq \mathrm{rt}(P_v) \\ \infty & \mathrm{type}(\iota_u^{-1}(v)) \neq \bot \wedge\ \iota_u^{-1}(v) \nsim \iota_v^{-1}(v) \\ 0 & \text{otherwise} \end{cases} \tag{5}$$

The first case deals with pattern borders. If an untyped node of a selected pattern is matched at v, then a root alternative must be selected there, because the pattern expects a register value as input at this point. The second case ensures consistency within a pattern, so all or no nodes of a pattern are replaced. If a typed node of a selected pattern is matched at v, then the corresponding node of an alternative pattern must be equivalent. Figure 5 gives a graphical example. The upper part shows three patterns Add, Add+Const and Const with their costs annotated respectively, which shall cover the program graph. The program graph in the lower left holds an empty node, which we will ignore in this example. Each pattern is matched once. Alternatives for Add and Const are generated in the u and v node, respectively. The Add+Const pattern spans two nodes, hence u and v have a corresponding alternative. The cover is displayed as a hatching instead of arrows like in Figure 2 for the sake of clarity. The costs of individual patterns are given externally and put into the cost vectors in the PBQP graph on the right. The cost model roughly shows the instruction size on IA32, so the PBQP solver will optimize for code size in this case.

Let us consider the cost matrix. The ∞ entry in the right column originates from the first case of Equation 5, because the Add pattern requires a root alternative at v, which is only given by the Const alternative. The ∞ entry in the left column comes from the second case of Equation 5, because the Const root alternative at v is incompatible with the Add+Const alternative in u. Thus, we ensure that a PBQP solution selects the Add+Const pattern in both nodes or not at all.

PBQP instances, constructed by this method, form a subclass of PBQP instances, which we call "graph of an instruction selection PBQP" (GISP).

4. ENSURING A PBQP SOLUTION

With the knowledge about PBQP-based instruction selection, we now treat the question whether a solution can be guaranteed. First we will show two problems of the classical method and how to solve them. Then verification is considered.

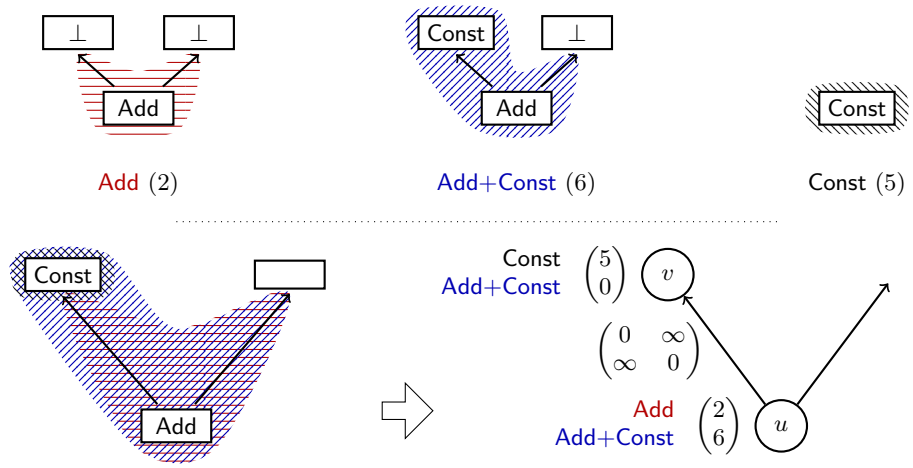


Figure 5: Mapping instruction selection to PBQP.

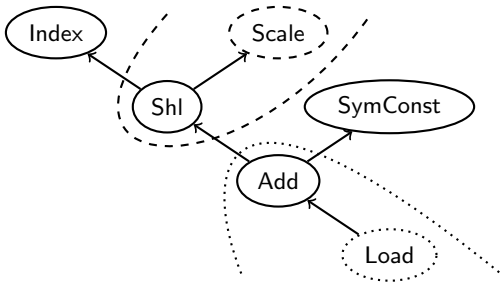


Figure 6: Example of conflicting decisions within a single pattern.

4.1 Conflicting decisions

First a quick note on the difference between the above formalisation and an implementation. In the formal reduction process alternatives are deleted, so vectors and matrices can "shrink", but for simplicity and efficiency reasons an implementation can mark a deleted alternative instead by assigning infinite costs. This is undesirable within the formalization, because the matrix normalisation would be unnecessarily complex. The first implementation of a PBQP solver [8, 9, 29] does not consider infinite vector costs and the resulting propagation effect of the normalisation. Thus, the PBQP solver may select alternatives which should have been deleted already. More precisely, one may think of a situation where a previous heuristic reduction has invalidated the entry with the lowest local cost, such that after selecting the entry the resulting PBQP instance has no solution. Without the propagation effect the local information, on which the heuristic reduction bases its decisions, is misleading.

Figure 6 shows such a scenario where the PBQP solving algorithm (a sufficient cost model postulated) selects the wrong alternative. First note that only a subgraph is shown and many more edges connected to the displayed nodes are possible. Thus, the PBQP solver may apply the heuristic RN at any time at any node. In the first step in our example the heuristic reduction chooses to exchange the dotted Load node for an ia32Mov node, whose pattern also includes all the other displayed nodes, especially the Scale node. Hence, the alternative containing the ia32Mov pattern must be selected at every other node, too. This fact is reflected at the Add node, where the other alternatives have infinite costs, but these costs are not propagated further, i.e. the costs are not propagated over the dotted boundary. Now a second heuristical decision at the dashed Scale node chooses an alternative not containing the ia32Mov pattern. This is inconsistent with the first heuristic decision, but the adjacent Shl does not indicate this invalidity and the heuristic only considers graph elements within the dashed boundary. With this unfortunate decision the PBQP instance does not have a solution anymore. Our instruction selector used the Eckstein-Scholz-PBQP solver[4] at first, but it could not compile certain programs due to the described problem.

Infinite costs must be assigned to the invalid entries in the Shl and Scale cost vectors to prevent the heuristic to select an incompatible entry in the second step. After the first heuristic reduction, the Add cost vector holds infinite cost entries, which must be propagated into the neighboring matrices and recursively into adjacent vectors. This infinity propagation in the implementation ensures the normalization of all edges affected by the previous reduction.

Before starting the PBQP solving algorithm and at the end of R2 and RN the InfinityPropagation procedure, shown in Algorithm 1, must be called for every neighbor of the reduced node. This extended algorithm solves the example problem above by guaranteeing that the following condition holds: If one alternative of a pattern match has infinite cost, so do all alternatives of this match. This is an extension to the classic solving algorithm, since InfinityPropagation propagates costs across the whole PBQP graph, instead of just into adjacent vectors. Additionally, independent edges may be removed during the infinity propagation, which could avoid some otherwise necessary heuristic decisions.

The time complexity of the PBQP solving algorithm is $O(nm^3)$ [29], where n is the size of the program graph and m the maximal size of the cost vectors. For a GISP m is the maximal number of equivalence classes $\max_{v \in V} |\mathcal{A}_v|$ of a node v. This upper bound of $O(nm^3)$ also holds for our infinity propagation extension, because every propagation step

[4] `http://www.complang.tuwien.ac.at/scholz/pbqp.html`

Algorithm 1 Infinity propagation procedure to embed into the PBQP solving algorithm [29].

```
1: procedure InfinityPropagation(x)
2:     for all y ∈ adj(x) do
3:         for i ← 1 to |c_x^→| do
4:             m ← min(C_xy(i, :))
5:             c_x(i) ← c_x(i) + m
6:             if m < ∞ then
7:                 a^→ ← C_xy(i, :) − m^→
8:                 C_xy(i, :) ← a^→
9:     S ← {i | 1 ≤ i ≤ |c_x^→| ∧ c_x(i) = ∞}
10:    for all y ∈ adj(x) do
11:        for all i ∈ S do
12:            C_xy(i, :) ← ∞^→
13:        normalize edge (x, y)
14:        if (x, y) is independent then
15:            remove edge (x, y)
16:        if C_xy changed then
17:            InfinityPropagation(y)
```

deletes at least one alternative, so nothing is done twice and the costs are amortized. Therefore, critical for the execution time is the number of alternatives, which is small and negligible in practice.

4.2 Overlapping patterns

The IR has a known, finite set of node types. Since the arity of each type specifies the number of successors, all possible atomic patterns can be enumerated. A pattern set which includes all atomic patterns is called *atomically complete*.

Atomic patterns can trivially cover any program graph, therefore at least one solution for the corresponding GISP exists. The number of possible solutions can only be reduced by reductions. Let us assess the effect of reductions on the solvability of GISPs.

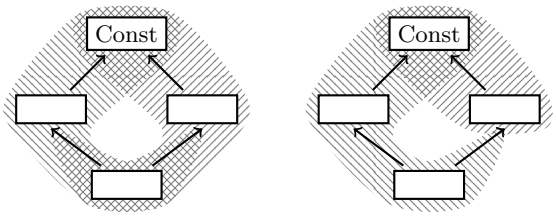


Figure 7: Critical GISP to motivate composability.

R1 and R2 sustain the solvableness and thus are uncritical for this problem. For RN on the other hand, additional constraints for the set of pattern graphs must hold, because being atomically complete is not sufficient for correct instruction selection. Figure 7 shows a problematic scenario on the left. The two symmetric patterns do *not* cover the program graph, because their roots do not belong to the same alternative. Therefore, selecting an alternative that induces their simultaneous selection, makes a solution impossible. Yet, RN could select such an alternative at the Const node without noticing the conflict, since only neighboring nodes are considered. After this selection, an atomic cover is not possible anymore, since the predecessors of the Const would not be compatible. The graph on the right displays a legal cover with two patterns. The smaller pattern is also part of the critical alternative at the Const node from above and makes it a legal selection. If this pattern would not exist, the two roots would have to be selected, which is impossible. If a pattern set also includes patterns for every part of every non-atomic pattern, this problem cannot occur. We call such a pattern set composable.

Definition 12 (Composable pattern set). *A pattern set $\mathcal{P}$ is called composable, if for every pattern graph $P \in \mathcal{P}$ and every node $v \in V_P$ with* $\mathrm{type}(v) \neq \bot$ *there is a cover C of P with*

$$\exists (P', \iota') \in C : \iota'(\mathrm{rt}(P')) = v \wedge P' \cong \mathrm{span}(v). \tag{6}$$

Furthermore, for the root $\mathrm{rt}(P)$ *of every pattern $P \in \mathcal{P}$ and for every successor node $w \in succ(\mathrm{rt}(P))$ a cover $\{(P_w, \iota_w)\}$ of* $\mathrm{rt}(P)$ *with $P_w \in \mathcal{P}$ must exist, where the following holds:*

$$\forall u \in succ(\mathrm{rt}(P)) \setminus \{w\} : \mathrm{span}(u) \cong \mathrm{span}(\iota_w^{-1}(u)) \tag{7}$$

and

$$\mathrm{type}(\iota_w^{-1}(w)) = \bot. \tag{8}$$

Definition 12 can be separated into two parts. On the one hand Equation 6 requires that every spanning graph within a pattern has a pattern itself. Figure 8 shows the non-tree pattern P_0 that resembles the ia32Inc instruction. The spanning graph of the Add node in the P_0 pattern requires P_1 and recursively P_3. The second part are the required Equation 7 and Equation 8, which state that any successor of the root node can be cut, in the sense that the subgraph is substituted by a $\bot$ node. In Figure 8, the spanning graph of the Add node in the P_0 pattern is substituted for $\bot$ in P_2. For all other nodes the pattern is equivalent, especially the inverse morphism ι^{-1} is defined for these nodes. Likewise, P_4 can be constructed by substituting the Load node in P_1. In this example, composability requires the existence of the atomic Store, Add and Load patterns. Untyped nodes are exempt. There is no atomic $\bot$ pattern.

4.3 Verification

We summarize the solutions to the two stated problems:

1. The PBQP solving algorithm must be extended with infinity propagation to identify conflicting entries as such.
2. The pattern set must be atomically complete and composable.

Employing these preconditions, we prove [2] that the instruction selection will always succeed.

While this theoretical proof is assuring, let us consider the practical applicability. In contrast to a manually programmed instruction selection, a code generator generator can additionally check the correctness of the specification, where "correctness" means that code generation will always succeed. The semantic equivalence of IR and generated code must be addressed at the specification and is assumed for our purposes. In other words the task is to find a cover for all possible program graphs and this means to statically check the two properties identified above:

Atomic Completeness: All possible atomic patterns can be enumerated and searched for in the rewrite rules. If an atomic pattern is not found, a detailed error message can be given indicating the missing pattern.

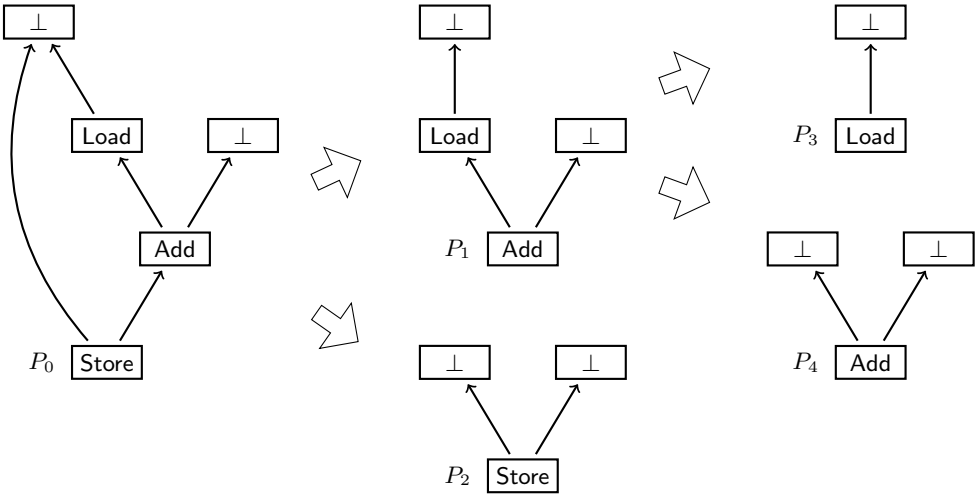
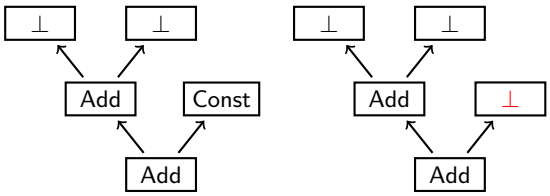


Figure 8: Implicated patterns for a composable pattern set.



(a) Pattern graph. (b) Missing pattern.

Figure 9: Pattern graph and missing pattern graph for a composable rule set.

Composability: (see also Definition 12) Given a complex pattern, we can deduce all subpatterns. For a complex rooted pattern any direct successor of the root node can be cut and substituted for a ⊥ node. The resulting pattern must be in the pattern set. Likewise, any spanning graph of a root successor must exist in the pattern set. We know the exact patterns in this case; if they lack, a detailed error message can be given.

4.4 Automatic rule generation

Not only can missing patterns be identified, but they can even be generated automatically, if the pattern set is atomically complete. The existing rule set can be used to find a cover of the missing pattern and compose a new rule from the rules used in the cover. The method is the same as performing instruction selection on program graphs. To ensure an optimal solution a brute-force algorithm could be used in this case, since pattern graphs are relatively small.

For example, the ia32Lea pattern graph of Figure 9(a) requires the pattern graph of Figure 9(b), otherwise the resulting rule set is not composable. For the missing pattern graph the generator computes the optimal cover consisting of two atomic Add pattern graphs. Then a new rule for the missing pattern graph is added to the rule set. Applying this rule is equivalent to applying the atomic Add rules twice. Therefore, the rule also costs twice as much.

We can conclude that specifying an atomically complete pattern set is sufficient to guarantee a composable generated pattern set. This in turn guarantees a successful PBQP-based instruction selection.

Bench.	GCC	libFirm	PBQP	libFirm/PBQP
gzip	187	182	186	97.7%
vpr	276	296	292	101.3%
gcc	123	130	135	96.5%
mcf	327	329	327	100.5%
crafty	135	124	134	92.5%
parser	257	261	264	98.8%
perlbmk	250	196	220	89.1%
gap	128	139	142	97.9%
vortex	200	206	214	96.3%
bzip2	236	242	244	99.2%
twolf	459	512	479	107.0%
Average				97.9%

Table 1: SPEC CINT2000 Comparison: PBQP-based vs. manual code generator. Benchmark execution times in seconds.

5. EVALUATION

We implemented our modified PBQP-based instruction selection within the libFirm compiler[5]. To evaluate the quality of the resulting code generated from our compiler, we used the SPEC CINT2000 benchmark suite.

Table 1 shows the execution times. We used the libFirm compiler with the conventional manually programmed instruction selection and with the PBQP-based one. To show that libFirm is competitive, we also included measurements of GCC 4.2.1. All test programs used the same optimization parameters and were tested on a Pentium 4 with 2.4Ghz and 1GB RAM.

The PBQP-based instruction selection produces code of slightly inferior quality, which can be seen at the numerous ratios of less than 100%. Possible reasons are

1. an insufficient use of the target architectures possibilities,

[5] `http://www.libfirm.org/`

Bench.	R1 + R2	RN	sub	$\frac{\text{RN}}{\text{R1+R2+RN}}$	$\frac{\text{sub}}{\text{R1+R2+RN}}$
gzip	7,131	0	0	0%	0%
vpr	22,940	42	0	0.137%	0%
gcc	277,967	52	2	0.012%	0.00048%
mcf	2,015	1	0	0.033%	0%
crafty	31,082	12	0	0.031%	0%
parser	28,637	0	0	0%	0%
perlbmk	114,786	3	0	0.002%	0%
gap	99,960	47	0	0.034%	0%
vortex	130,298	9	0	0.005%	0%
bzip2	7,189	2	0	0.019%	0%
twolf	40,140	32	0	0.065%	0%
	762,145	200	2	0.018%	0.00018%

Table 2: Reduction counts of CINT2000 programs.

2. an unrealistic cost model,
3. a weak pattern language compared to manual programming or
4. heuristical PBQP decisions, which produce a suboptimal result.

During the development of the rule set, we observed that additional rules, which exploited additional possibilities of the target architecture, improved the code quality tremendously, while modifications of the cost model did not result in significant speed improvements. Regarding the pattern language, we used the generic graph transformation tool GrGen [15] in our prototype, which offers all desired features. Heuristic PBQP decisions after all demand more attention.

To investigate how close to the optimum the PBQP results are, we implemented an alternative solving algorithm, which uses brute force instead of heuristic reductions. Despite being surprisingly fast in our experiments, this approach does not scale.

We compare heuristic with brute force decisions to identify suboptimal ones. Table 2 shows the suboptimal reductions count in the sub column. Only 2 out of 200 heuristic reductions chose a suboptimal alternative. Furthermore, heuristic decisions are so rare (0.018%) compared to optimum-preserving reductions that 99,99982% of all nodes are optimally reduced. Therefore, the heuristic decisions seem not to be the reason for the observed inferior code quality.

6. RELATED WORK

The PBQP was first used for register allocation [17, 29] and is often [20, 25, 31] classified with approaches like integer linear programming, since it can be used to find an optimal solution. However, using heuristics the PBQP can still be solved in linear time.

Later the PBQP was also applied to instruction selection [8, 9, 10]. While previous approaches were restricted to trees [23] and can be extended to DAG-shaped program graphs [13], the PBQP approach is more flexible as it also allows patterns to be graphs. Various extensions [7, 19, 30] show this flexibility by using DAG-shaped patterns to implement instructions that produce multiple results. Also rematerialization was introduced [19], which means selecting multiple alternatives at once. These extensions are desirable for irregular architectures and custom instructions [21] common in embedded systems. Our formalization provides foundations to verify the correctness of these extensions.

7. FUTURE WORK

Current graph transformation tools are not optimized to match lots of overlapping patterns. For a fast graph-based instruction selection further research must either extend tree pattern matchers or speed up graph pattern matchers in this special case. Since pattern matching is a separate problem, inspiration may come from various sources like subtree enumeration [1], BURS [24] or RETE [3].

Another remaining problem is to verify the various extensions of the PBQP approach mentioned in the previous section. The challenge is to find appropriate preconditions for the verification. Additionally, an interesting idea is to refine these preconditions by employing knowledge about the IR and the pattern set.

8. CONCLUSION

In this work we developed a formal foundation for graph-based instruction selection, which is necessary to verify correctness. Using our formalization we identified and resolved two problems with the known PBQP-based approach, which highlights the need for verification and formalization.

Automatic rule generation eases the architecture specification process. Our benchmark results show that a PBQP-based approach yields code quality comparative to manually fine-tuned instruction selections. Due to these two reasons we believe that instruction selection should not be restricted to tree shapes anymore. Also the execution times during the development improved nearly exclusively because of additional patterns. Hence, the most effective way to reduce execution times seem to be more replacement options. This also motivates to lift instruction selection from tree to graph patterns.

9. ACKNOWLEDGMENTS

We would like to thank Matthias Braun, Jürgen Graf, Martin Hecker, Denis Lohner, Christoph Mallon, Daniel Wasserrab and all anonymous reviewers for their helpful comments.

10. REFERENCES

[1] P. Biswas and G. Venkataramani. Comprehensive isomorphic subtree enumeration. In *CASES '08: Proceedings of the 2008 international conference on Compilers, architectures and synthesis for embedded systems*, pages 177–186, New York, NY, USA, 2008. ACM.

[2] S. Buchwald and A. Zwinkau. Befehlsauswahl auf expliziten Abhängigkeitsgraphen. Master's thesis, Universität Karlsruhe (TH), IPD Goos, December 2008.

[3] H. Bunke, T. Glauser, and T. Tran. An efficient implementation of graph grammars based on the RETE matching algorithm. In H. Ehrig, H.-J. Kreowski, and G. Rozenberg, editors, *Graph Grammars and Their Application to Computer Science*, volume 532 of *Lecture Notes in Computer Science*, chapter 17, pages 174–189. Springer-Verlag, Berlin/Heidelberg, 1991.

[4] C. N. Click. *Combining Analyses, Combining Optimizations*. PhD thesis, Rice University, February 1995.

[5] G. Cohen, J.-P. Quadrat, G. J. Olsder, and F. Baccelli. Synchronization and linearity, an algebra for discrete event systems, 1992.

[6] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, October 1991.

[7] D. Ebner, F. Brandner, B. Scholz, A. Krall, P. Wiedermann, and A. Kadlec. Generalized instruction selection using SSA-graphs. In *LCTES '08: Proceedings of the 2008 ACM SIGPLAN-SIGBED conference on Languages, compilers, and tools for embedded systems*, pages 31–40, New York, NY, USA, 2008. ACM.

[8] E. Eckstein. *Code optimizations for digital signal processors*. PhD thesis, TU Wien, November 2003.

[9] E. Eckstein, O. König, and B. Scholz. Code instruction selection based on SSA-graphs. In *SCOPES*, pages 49–65, 2003.

[10] E. Eckstein and B. Scholz. Addressing mode selection. In *CGO '03: Proceedings of the international symposium on Code generation and optimization*, pages 337–346, Washington, DC, USA, 2003. IEEE Computer Society.

[11] H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. *Fundamentals of Algebraic Graph Transformation (Monographs in Theoretical Computer Science. An EATCS Series)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[12] S. Eilenberg and S. MacLane. General theory of natural equivalences. *Transactions of the American Mathematical Society*, 58(2):231–294, 1945.

[13] M. A. Ertl. Optimal code selection in DAGs. In *POPL '99: Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 242–249, New York, NY, USA, 1999. ACM.

[14] C. W. Fraser, R. R. Henry, and T. A. Proebsting. Burg: fast optimal instruction selection and tree parsing. *SIGPLAN Not.*, 27(4):68–76, 1992.

[15] R. Geiß, G. V. Batz, D. Grund, S. Hack, and A. M. Szalkowski. GrGen: A fast SPO-based graph writing tool. In A. Corradini, H. Ehrig, U. Montanari, L. Ribeiro, and G. Rozenberg, editors, *Graph Transformations - ICGT 2006*, Lecture Notes in Computer Science, pages 383 – 397. Springer, 2006. Natal, Brasil.

[16] A. Habel, R. Heckel, and G. Taentzer. Graph grammars with negative application conditions. *Fundam. Inf.*, 26(3-4):287–313, 1996.

[17] L. Hames and B. Scholz. Nearly optimal register allocation with PBQP. In D. E. Lightfoot and C. A. Szyperski, editors, *Modular Programming Languages, 7th Joint Modular Languages Conference, JMLC 2006, Oxford, UK, September 13-15, 2006, Proceedings*, volume 4228 of *Lecture Notes in Computer Science*, pages 346–361. Springer, 2006.

[18] R. Heckel. Graph transformation in a nutshell. *Electronic Notes in Theoretical Computer Science*, 148(1):187–198, February 2006.

[19] H. Jakschitsch. Befehlsauswahl auf SSA-Graphen. Master's thesis, IPD Goos, November 2004.

[20] D. Koes and S. C. Goldstein. A progressive register allocator for irregular architectures. In *CGO '05: Proceedings of the international symposium on Code generation and optimization*, pages 269–280, Washington, DC, USA, 2005. IEEE Computer Society.

[21] T. Li, Z. Sun, W. Jigang, and X. Lu. Fast enumeration of maximal valid subgraphs for custom-instruction identification. In *CASES '09: Proceedings of the 2009 international conference on Compilers, architecture, and synthesis for embedded systems*, pages 29–36, New York, NY, USA, 2009. ACM.

[22] M. Löwe and H. Ehrig. Algebraic approach to graph transformation based on single pushout derivations. In *WG '90: Proceedings of the 16th international workshop on Graph-theoretic concepts in computer science*, pages 338–353, New York, NY, USA, 1991. Springer-Verlag New York, Inc.

[23] A. Nymeyer and J.-P. Katoen. Code generation based on formal burs theory and heuristic search. *Acta Informatica*, 34(8):597–635, 1997.

[24] E. Pelegrí-Llopart and S. L. Graham. Optimal code generation for expression trees: an application BURS theory. In *POPL '88: Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 294–308, New York, NY, USA, 1988. ACM.

[25] F. M. Q. Pereira and J. Palsberg. Register allocation by puzzle solving. *SIGPLAN Not.*, 43(6):216–226, 2008.

[26] T. Proebsting. Least-cost instruction selection in DAGs is NP-complete. Privately published online, 1998.

[27] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global value numbers and redundant computations. In *POPL '88: Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 12–27, New York, NY, USA, 1988. ACM.

[28] G. Rozenberg, editor. *Handbook of graph grammars and computing by graph transformation: volume I. foundations*. World Scientific Publishing Co., Inc., River Edge, NJ, USA, 1997.

[29] B. Scholz and E. Eckstein. Register allocation for irregular architectures. In *LCTES-SCOPES*, pages 139–148. ACM, 2002.

[30] A. Schösser. Graphersetzungsregelgewinnung aus Hochsprachen und deren Anwendung. Master's thesis, Universität Karlsruhe (TH), IPD, 9 2007.

[31] M. D. Smith, N. Ramsey, and G. Holloway. A generalized algorithm for graph-coloring register allocation. In *PLDI '04: Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, pages 277–288, New York, NY, USA, 2004. ACM.