210: Compiler Design

Spring 2018

Homework 5

AST Optimizations: May 22, 2018, 10:00
Final Submission: June 1, 2018, 23:59

50 Points

Introduction

The objective of this homework is to implement optimizations in the JavaLi compiler. The goal of optimizations is to minimize the resources that are required during the execution of a program. Although there exist various optimizations, e.g., minimization of memory requirements, the focus of this homework is to improve the performance of a program in terms of execution time. Particularly, your task is to implement optimizations that make programs run both correctly and faster.

Description

You should implement a number of optimizations that reduce the number of assembly instructions required for the execution of a program. You are free to choose any optimizations you want and as many as you want. The goal is to make JavaLi programs run as fast as possible. The optimizations may modify either the Abstract Syntax Tree (AST) or the generated code, i.e., assembly instructions.

Most optimizations require information gathered by data-flow analyses, and the implementation of data-flow analyses relies on the control-flow graph (CFG). To simplify your task and allow you to focus on the implementation of optimizations, we provide the implementation of the CFG and an abstract class for data-flow analyses that you should extend depending on your needs.

Control-Flow Graph The construction of the CFG is placed into a separate phase of the compiler. This phase executes after the semantic analysis. The CFG is built for each method using the class `cd.transform.CfgBuilder`. The control-flow graph is stored in a new data structure, which is separate from the AST. The classes `ControlFlowGraph` and `BasicBlock` in the `cd.ir` package are used to represent control-flow graphs. A single instance of type `ControlFlowGraph` is associated with each `MethodDecl` instance (a new field `cfg` has been added to `MethodDecl` for this purpose). `BasicBlock`s consist of a sequence of `Stmt`s and possibly of a condition. Note that AST nodes representing control flow, such as `IfElse` or `WhileLoop`, are not placed into a `BasicBlock`'s list of instructions. Instead, their control flow is reflected in the structure of the CFG. In addition, blocks are connected to each other, using the `successors` and `predecessors` fields.

Data-Flow Analysis All data-flow analyses share a common property: they use a fixed-point iteration scheme to approximate the program states. Since your optimizations may need the implementation of one or more data-flow analyses, we provide the `DataFlowAnalysis` abstract class in the `cd.transform.analysis` package. The method `iterate` performs forward flow fixed-point iteration until there is no change. If you need a backward flow analysis, you should properly extend the provided implementation. To implement a concrete data-flow analysis, you should implement a subclass of the `DataFlowAnalysis` class, and provide implementations for the methods `initialState()`, `startState()`, `transferFunction()`, and the method `join()`. As presented in the lecture, data-flow analysis can be implemented efficiently by computing the *gen* and *kill* sets for each basic block. This should be done in the constructor of the concrete data-flow analysis, before calling `iterate()`, such that these sets are ready when the `transferFunction()` is called.

Task 1 (Due: May 22, 2018)

You are required to perform a number of optimizations that modify the AST, e.g., constant folding. Since the most important requirement of every optimization is correctness and then performance, we highly recommend you to start with simple optimizations and then try to apply some more interesting ideas. For example, you can start with intra-procedural analysis and later you can work on inter-procedural analysis. Moreover, you can start with optimizations about local scalar variables and then you can handle fields and arrays. There is no minimum requirement for the number of implemented optimizations, but for this task, only AST optimizations will be considered.

Task 2 (Due: June 1, 2018)

For this task, you are allowed to perform any kind of optimization that you think may minimize the execution time of a program, i.e., AST optimizations and direct optimizations for assembly instructions, e.g., redundant null check elimination. Therefore, all optimizations will be considered for this task.

Testing

We provide a number of tests, but, as usual, you are required to write additional tests that show the performance benefits of your optimizations. The JavaLi compiler should work when the optimizations are enabled, but you should ensure that it also works when the optimizations are disabled.

Grading

You should implement at least one optimization that modifies the Abstract Syntax Tree (AST) for the first task. Any optimization during code generation will be ignored for the first task. For the second task, you are allowed to add more AST optimizations and/or perform optimizations on assembly instructions level. The grade for your implementation will be determined automatically based on the performance of the generated code for a set of tests; correctness is a fundamental requirement. We will not grade each optimization independently. Hence, you should think about optimizations that have significant impact on most programs. We will measure the number of executed instructions, i.e., AST instructions for task 1 and assembly instructions for task 2.

Hand-in

The final submission should be committed into your team's repository by the specified deadline. Any commit after the deadline will not be considered. In addition, you should include any tests you created, in a subdirectory of `javali_tests` named `HW5_team`. Finally, you should provide a `README` file in the `HW5` directory of your team's repository. You should describe there the optimizations that you implemented and any special cases that your implementation may not handle.