Summary of
Finite-State Code Generation

This is a summary of the paper “Finite-State Code Generation” by Christopher W. Fraser and Todd A. Proebsting. The paper describes GBURG, a “code generator generator.” The GBURG generated generators/translators use a finite state pattern matcher to translate the input intermediate code (LVM code) into target machine instructions.

1 GBURG

The input to GBURG is a machine description. It consists of an instruction grammar annotated with C code for emitting target instructions. GBURG translates the machine description into a C code function `compile()` that translates LVM code into target code. The function is actually a hard-coded pattern matcher that is optimized for speed and size.

Because of the simple matching technique, the descriptive power of the machine description grammar is that of regular expressions.

2 Lean Virtual Machine

The input intermediate code that GBURG generated translators use is called LVM, “Lean Virtual Machine.” It is based on the intermediate representation in lcc, but converted into stack machine instructions.

The LVM has registers, memory, and a minimal (“lean”) instruction set. For example, load and store instructions always find the memory address on the stack.

The instruction set does not assume any particular source language, but every language construct can be mapped to a sequence of LVM instructions.

3 Left-Bias

The LVM code is specially designed to solve some of the problems of the instruction selection method. The linear view of the program prevents the translator from knowing what (binary) operation it eventually encounters when it is looking at the left operand\(^1\). Instead, a choice must be made that will work whatever the operator is, which usually means that the left operand has to be put in a register. When the right operand is translated, the next input instruction is the operator and a better instruction choice can be made.

This “left-bias” cannot be removed since this would require an explicit tree representation of the program. The whole point of GBURG is to avoid trees and to use a simple linear representation. However, the problem can be mitigated by careful design of the input code (LVM code). If one

\(^1\)Remember structure of stack code: `compute_left_op compute_right_op operator`
of the operands can be used for generating special cases of the operator instruction, this operand should be the right operand. Examples of special cases are addressing modes for load and store instructions (the translator can recognize a short sequence that ends with a load and a store and generate special code if this sequence matches an addressing mode), and special forms of arithmetic instructions (if the right operand is a constant, a special “use immediate constant” variant of the operator instruction can be used).

4 Results

The translators are small and fast since gb Burg uses several methods to reduce their size. Other code generators that use more or less tree-ish representation internally such as those generated by burg and iburg are outperformed in translation speed (factor 2–3) and code generator size (factor 3–4). The code quality is not compared, it is only stated that it “suffer very little” from the more restricted instruction selection technique.

The paper compares execution time for a set of benchmark programs. The programs were compiled with two ordinary compilers, lcc and MSVC with maximum optimizations. They were also translated from LVM code into executable code by a GBURG generated translator. The execution time of the result was measured, and in the GBURG case, the translation time was included in the execution time.

The execution time of the code from the GBURG generated translator is within factor two of that of lcc and within factor four of that of MSVC. This means that a GBURG generated translator with its small size and high translation speed could be a suitable alternative to a simple interpreter for a small system. Interpreters usually execute code with a slowdown of factor 10 compared to compiled code.