Grammar-based Compression of Interpreted Code

William S. Evans and Christopher W. Fraser

Programs that must run at or near top speed must use native machine code, but some programs have more modest performance requirements. For example, a cellular telephone handset might include software that reacts to keystrokes and spends most of its time waiting for the next input. An interpreted bytecode can run fast enough for such applications and can be smaller than machine code. The freedom to cater a program encoding for size offers additional opportunities (and challenges) for program compaction. This paper describes compact encodings for interpreted languages.

Good encodings are hard to design. They must anticipate the sequences of operations that programmers will use most often and assign the shortest codes to the most common sequences.

Typical programs can be analyzed for statistics to guide the design. Indeed, the designer of a compact representation may target a single program and design a compact language exclusively for that program. Of course, designing a language for every program is too labor intensive to be done by hand. It requires automation. Also, it requires a different interpreter for each compacted program, which can also be expensive. A better solution may be to design an interpreter for a set of programs and use one interpreted language for all.

This paper focuses on the automatic design and implementation of compact interpretable bytecodes. The objective is a form that is compact for a set of sample programs and for other programs with similar characteristics. The key to designing such compact bytecodes is to identify frequently occurring patterns of program constructs, and to replace them with a single interpreted construct. This process is unlike the Huffman fixed-to-variable length coding, which encodes single symbols using a variable number of bits, and more like Tunstall variable-to-fixed length coding, which encodes multiple symbols as a single, fixed size codeword [Tunstall].

Representation

We could start our search for frequently occurring patterns with programs represented in a high-level language (e.g. C++), an intermediate code (e.g. bytecode or expression trees), or the instruction set of a target machine. Since our goals include direct interpretation, starting with a high-level language can be problematic. Few high-level languages are directly interpreted, so our compacting interpreter would itself need to produce a lower level representation for interpretation. At the other end of the spectrum, we may start with machine code and have the interpreter translate its compact representation into instructions that can be directly executed. It can, however, be tricky for the interpreter to maintain control of the execution of the program in this case. Thus most systems rely on a compiler front end to produce an intermediate code that can be interpreted.

Some systems create specialized instructions for common sequences in postfix bytecode [Clausen et al; Latendresse]. Others operate on the corresponding expression trees [Hoogerbrugge et al; Proebsting]. For example, a simple expression, such as `1+(2×3)`, translates into the tree `AddInt(1, MulInt(2, 3))`. Proebsting's greedy heuristic looks for the most frequent parent/child pair in all the expression trees and creates a new instruction or "superoperator" for that pair of operations. For example, if multiplication by two is the most common pair, then a new, unary instruction `MulInt(2,*)` replaces all multiplications by two. After replacement, our example expression would use only two operands and two operations, rather than the original three operands and two operations. This process may be

repeated to obtain a desired interpreter language size. Ernst et al. describe a similar method that enhances a RISC-like virtual machine instruction set [Ernst].

A variation on the superoperator theme represents the program as a *dictionary* and a *skeleton* [Liao]. The dictionary contains the enhanced instructions while the skeleton consists of a sequence of original machine code instructions and references (calls) to the enhanced instructions in the dictionary. The skeleton, in this case, acts as an interpreter.

The methods above and the one described here [Evans and Fraser] all use addressable codes, typically bytecodes. A notable alternative adapts Huffman-coded instruction sets for direct interpretation. Methods have been devised to minimize the cost of bit-level operations in such interpreters [Choueka et al; Latendresse].

Grammar-based Techniques

Programming languages normally obey a grammar, and this restriction can help compression. We needn't represent invalid programs, which confers an immediate advantage over general-purpose compression schemes, which must compress everything. A grammar also categorizes program elements, and compressors can exploit this labeling by specializing an encoding for each element type.

Thompson and Booth describe how to use a probabilistic grammar for a context-free language to encode strings from the language [Thompson and Booth]. One of their techniques, termed *derivation encoding* by Stone [Stone], represents a program by the sequence of grammar rules used to derive it (in a leftmost or rightmost derivation) from the starting symbol of the grammar. Thompson and Booth suggest using a Huffman code, based on the probabilities of the grammar rules, to encode the rule choices.

Another grammar-based encoding method, termed *parsing encoding* by Stone, represents a program as the sequence of choices made by a parser as it processes the program. A top-down parser makes essentially the same choices as the derivation encoding, but a bottom-up or shift-reduce parser is different. The parser is typically a pushdown automaton, and the choices it makes are which action (shift or reduce) to perform and which state to transition to.

Cameron [Cameron] demonstrated the power of derivation encoding by using a probabilistic grammar to obtain a derivation along with its probability. He then encoded the derivation using an arithmetic encoder and was able to compress programs to almost 10% of their original size.

These methods do not produce interpretable results. The compressed form of the program must be decompressed and compiled before execution.

Grammar Rewriting

How can we exploit the compression potential of grammar-based methods in a language that an interpreter can decode without decompressing it first? One answer [Evans and Fraser] starts with some representative sample programs and a grammar for the original (uncompressed) instruction set. Instructions in a program correspond to rules in the grammar and thus to steps in a parser's derivation for the program. The compressor transforms the grammar so that it parses the same language but uses fewer derivation steps, at least for the sample code. The revised grammar defines a bytecode that will be smaller for the sample and for similar programs. An example of this is given in Figure 1.

The derivation is a list of the rules used to expand the leftmost non-terminal in each sentential form of the derivation. Each rule is represented as an index: the *i*th rule for a non-terminal is represented as the index *i*.

Note that a separate derivation is generated for each basic block. Keeping the derivations separate allows direct interpretation of this representation. When

the interpreter encounters a control transfer, it knows that the derivation sequence at the target starts a derivation from the start non-terminal and applies a rule with the start non-terminal on the left-hand side.

Each rule number, in conjunction with the current non-terminal, acts as an interpreter instruction. We can see that for this grammar there are three instructions: 0, 1, and 2. Unless we encode each rule number as a byte, this is not, in general, a very practical code for interpretation. In order to create a practical and concise encoding of the program, we modify the grammar so that each non-terminal has close to 256 rules. The modification process takes two rules, $A \rightarrow \alpha\ B\ \beta$ and $B \rightarrow \gamma$, and adds to the grammar a third rule, $A \rightarrow \alpha\ \gamma\ \beta$, where A and B are non-terminals and α, β, and γ are strings of terminals and non-terminals. We call this process *inlining* a B rule into an A rule. Inlining doesn't change the language accepted by the grammar. However, it shortens the sequence of rules (the derivation) needed to express some strings, and it increases the number of rules for some non-terminal.

Which rules should we inline? The goal of the inlining is to produce a grammar that provides short derivations for programs. Starting with a derivation of a program using the original grammar, the best single inline that we could perform is the most frequently occurring pair of rules; one used to expand a non-terminal on the right-hand side of the other. If this pair were used m times in the derivation, inlining would decrease the derivation length by m rules.

We can view this process as operating on the forest of parse trees obtained from parsing the original, uncompressed sample program using the original grammar. The parse produces a forest since we restart the parser from the start non-terminal at every potential branch target (i.e. `Label`). For our purposes, a parse tree is a rooted tree in which each internal node is labeled with a rule and each leaf with a terminal symbol. The root is labeled with a rule for the start non-terminal. In general, an internal node that is labeled with a rule $A \rightarrow a_1\ a_2\ ...\ a_k$ (where a_i is a terminal or non-terminal symbol) has k children. If a_i is a non-terminal then the ith child (from the left) is labeled with a rule for non-terminal a_i. If a_i is a terminal then the ith child is a leaf labeled with a_i. The program appears as the sequence of labels at the leaves of the parse trees in the forest, reading from left to right. A leftmost derivation is simply the sequence of rules encountered in a preorder traversal of each parse tree in the forest.

The inlining of one rule r_B into another rule r_A creates a new rule r'_A whose addition to the grammar permits a different (and shorter) parse of the program. One such new parse can be obtained by *contracting* every edge from a node labeled r_A to a node labeled r_B in the original forest – meaning the children of r_B become the children of r_A – and relabeling the node labeled r_A with the new rule r'_A. See Figure 2. If the number of edge contractions is m, the resulting forest has m fewer internal nodes and thus represents a derivation that is shorter by m steps.

To construct an expanded grammar, we parse a sample program (or a set of sample programs) using the original grammar and obtain a forest of parse trees. We then inline the pair of rules at the endpoints of the most frequent edge in the forest, contract all occurrences of this edge, add the new inlined rule to the grammar, and repeat. We stop creating rules for a non-terminal once it has 256 rules and thus create one bytecoded instruction set for each non-terminal. A grammar with several non-terminals will thus result in programs that interleave bytecodes for several different non-terminals, but the interpreter always knows how to decode the next byte because the context defines the non-terminal used to parse the next byte.

Occasionally, a rule for a non-terminal may be subsumed by a new rule. That is, after the addition of the new rule, the first rule is no longer used in the derivation. If the unused

rule is one that was added via inlining, we are free to remove it from the grammar. (We cannot, however, remove any of the original grammar rules, or we risk changing the grammar's language.) In our current implementation, we remove unused inlined rules in order to decrease the size of the expanded grammar. This removal may cause some non-terminals to have fewer than 256 rules. The implementation could be made to respond with more inlining, but the number of reductions is typically small, and the incremental value of the next inlining step drops with time, so the added complexity might not pay off.

This construction procedure is greedy; it always inlines the most frequent pair of rules. This is a heuristic solution to the problem of finding a set of rules to add to the grammar that permits the shortest derivation of the sample program. We rely on this heuristic since finding an exact solution is, unfortunately, NP-hard.

The resulting expanded grammar is ambiguous, since we leave the original rules in the grammar. We can use any valid derivation, but the size of the representation is the number of rules in the derivation, so compression demands a minimum length derivation. We use Earley's parsing algorithm, slightly modified, to obtain a shortest derivation for a given sequence. The derivation is then the compressed bytecode representation of the program and is suitable for interpretation.

The interpreter

This system has two interpreters. The initial interpreter accepts the initial, uncompressed bytecode. The initial interpreter and the expanded grammar form the raw material from which the system builds the second interpreter, which accepts compressed bytecode.

At the core of the initial interpreter is a routine comprised of a single C switch:

```
void interpret1(
  unsigned char op, istate *istate
) {
  switch (op) { … }
}
```

The routine accepts a single, uncompressed operator and pointer to a structure that records the state of an interpreter. The interpreter state could be maintained as variables local to a single interpretation routine, but it was helpful to be able to change the state from multiple routines.

The switch above has one case for each instruction in the initial instruction set, and the cases manipulate a small execution stack. Stack elements use a union of the basic machine types. For example, the case for `AddInt` pops two elements, adds them as integers, and pushes the result:

```
case AddInt:
stack = istate->stack;
a = stack[istate->top--].i;
b = stack[istate->top--].i;
stack[++istate->top].i = a + b;
return;
```

The base interpreter, which is called interp, simply calls interpret1 repeatedly. The second interpreter, which interprets compressed bytecodes, introduces another level of interpretation between `interp` and `interpret1`:

```
void interp(istate *istate) {
  while (1)
    interpNT(istate, NT_start);
}
```

`InterpNT` adds an argument that identifies a non-terminal and thus which of several specialized bytecoded instruction sets to use. `InterpNT` fetches the next bytecode, which, with the given non-terminal, identifies the rule for the next derivation step. A table encodes for each rule the sequence of terminals and non-terminals on the rule's right-hand side. `InterpNT` advances left-to-right across this right-hand side. When it encounters a terminal symbol, it calls `interpret1` to execute that symbol. When it encounters a non-terminal, it calls itself recursively, with the given non-terminal to define the new context and new specialized bytecode.

Performance

Table 1 below reports the size of several bytecode sequences as compressed by our

method. Each input was compressed twice, with grammars generated from two different training sets, namely the compilers `lcc` and `gcc`. Predictably, `lcc` and `gcc` each compress somewhat better with their own grammar, but the other inputs compress about as well with either grammar.

The interpreters are small: 4,029 bytes for the initial, uncompressed bytecode and 13,185 for the bytecode generated from the `lcc` training set. Thus adding 9,156 bytes to the interpreter saves roughly 900KB in the bytecode for `gcc`. The grammar occupies 8,751 bytes and thus accounts for most of the difference in interpreter size.

The initial, uncompressed bytecode takes roughly 70 times longer than native machine code, and the compressed bytecode adds another factor of two, but trade-offs favored size at every turn. Interpreter state is stored on the stack to simplify implementation, but it could be moved to registers. Also, double interpretation could be eliminated by hard-coding a switch for the compressed bytecode, which would suit systems that burn the interpreter into a cheap ROM but that download bytecode into scarcer RAM.

For calibration and as a very rough bound on what might be achievable with good, general-purpose data compression, gzip compresses the original, uncompressed inputs above to 31-44% of their original size, with the larger inputs naturally getting the better ratios. Thus the compressed bytecode is competitive with gzip despite operating with an additional constraint, namely support for direct interpretation's random access. For example, gzip is free to exploit redundant patterns that span basic blocks, where our bytecode compressor must stop and discard all contextual information at every branch target.

References

1. R. D. Cameron. Source encoding using syntactic information models. *IEEE Transactions on Information Theory 34,* 4 (1988), 843-850.
2. Y. Choueka, S. T. Klein, and Y. Perl. Efficient variants of Huffman codes in high level languages. *Proceedings of the 8th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval* (1985), 122-130.
3. L. Clausen, U. Schultz, C. Consel, and G. Muller. Java bytecode compression for low-end embedded systems. *ACM TOPLAS 22*, 3 (May 2000), 471-489.
4. J. Ernst, W. Evans, C. W. Fraser, S. Lucco, and T. A. Proebsting. Code compression. *Proceedings of the ACM SIGPLAN'97 Conference on Programming Language Design and Implementation*, 358-365.
5. W. Evans and C. W. Fraser. Bytecode compression via profiled grammar rewriting. *Proceedings of the ACM SIGPLAN'01 Conference on Programming Language Design and Implementation*, 148-155.
6. J. Hoogerbrugge, L. Augusteijn, J. Trum, and R. van de Weil. A code compression system based on pipelined interpreters. *Software-Practice and Experience 29*, 11 (1999), 1005-1023.
7. M. Latendresse. Automatic generation of compact programs and virtual machines for Scheme. *Proceedings of the Workshop on Scheme and Functional Programming 2000*, 45-52.
8. S. Liao, S. Devadas, and K. Keutzer. A text-compression-based method for code size minimization in embedded systems. *ACM Transactions on Design Automation of Electronic Systems 4,* 1 (Jan. 1999), 12-38.
9. T. A. Proebsting. Optimizing an ANSI C interpreter with superoperators. *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Jan. 1995), 322-332.
10. R. G. Stone, On the choice of grammar and parser for the compact analytical encoding of programs. *Computer Journal 29*, 4 (1986), 307-314.
11. R. A. Thompson and T. L. Booth. Encoding of probabilistic context-free languages. In Z. Kohavi and A. Paz, editors, *Theory of Machines and Computations*, 169-186. Academic Press, 1971.
12. B. P. Tunstall. *Synthesis of Noiseless Compression Codes*. Ph.D. dissertation, Georgia Inst. Technology, 1967.

C-code

```
void check(int flag) {
  if (flag == 0)
    exit( 0 );
}
```

Interpretable code

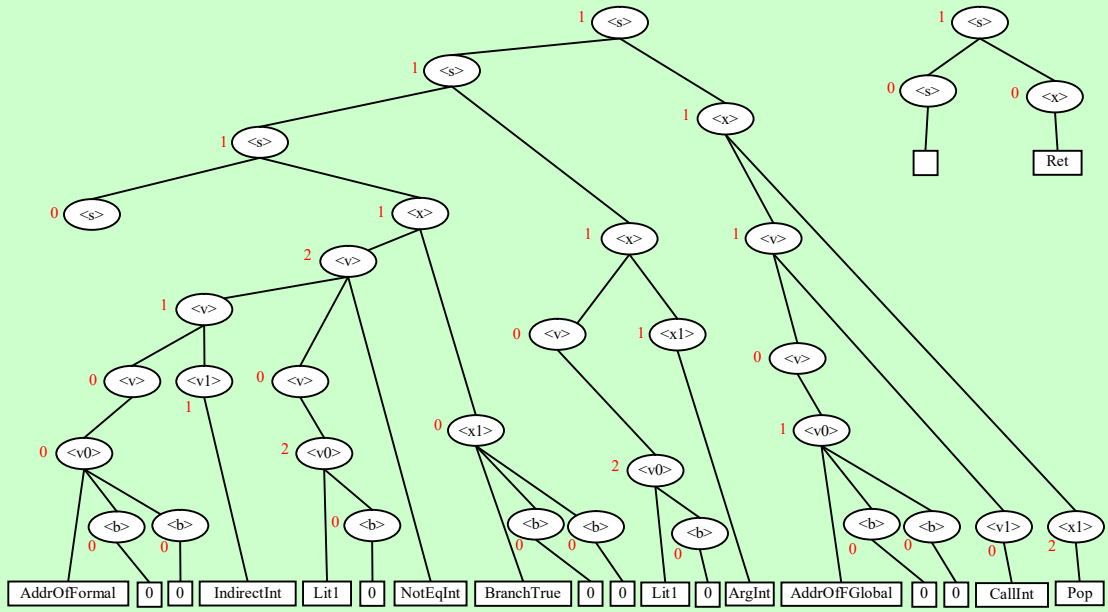
```
Entry: AddrOfFormal 0 0
       IndirectInt Lit1 0
       NotEqInt BranchTrue 0 0
       Lit1 0 ArgInt
       AddrOfGlobal 0 0
       CallInt
       Pop

Label: Ret
```

Grammar

```
0. <s> =
1. <s> = <s> <x>
0. <x> = Ret
1. <x> = <v> <x1>
0. <v> = <v0>
1. <v> = <v> <v1>
2. <v> = <v> <v> NotEqInt
0. <v1> = CallInt
1. <v1> = IndirectInt
```

```
0. <v0> = AddrOfFormal <b> <b>
1. <v0> = AddrOfGlobal <b> <b>
2. <v0> = Lit1 <b>
0. <x1> = BranchTrue <b> <b>
1. <x1> = ArgInt
2. <x1> = Pop
0. <byte> = 0
```

Derivation



Encoding

```
1 1 1 0 1 2 1 0 0 0 0 1 0 2 0 0 0 0 1 0 2 0 1 1 1 0 1 0 0 0 2
1 0 0
```

Figure 1: Using a grammar to encode interpretable code.
A compiler translates the C code in the upper left corner into the interpretable code to its right. One derivation of the interpretable code using the given grammar is shown as a parse tree. For each non-terminal, the grammar rules are numbered. The encoding is the sequence of rules used in a leftmost derivation. Note that each extended basic block has a separate derivation.

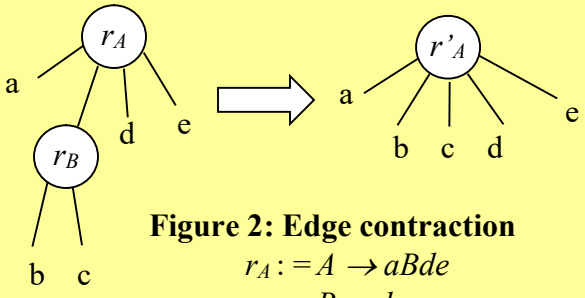


Figure 2: Edge contraction

$r_A := A \rightarrow aBde$

$r_B := B \rightarrow bc$

$r'_A := A \rightarrow abcde$

input	*original*	*compressed*			
		trained on gcc		*trained on lcc*	
		bytes	*ratio*	*bytes*	*ratio*
`gcc`	1,423,370	471,111	33%	577,814	41%
`lcc`	199,497	75,077	38%	57,722	29%
`gzip`	47,066	19,466	41%	19,706	42%
`8q`	436	138	32%	152	35%

Table 1: Benchmark program sizes before and after compaction