A PROGRAM EXECUTION MODEL BASED ON GENERATIVE DYNAMIC GRAMMARS

John Aycock
Department of Computer Science
University of Calgary
Calgary, Alberta, Canada
email: aycock@cpsc.ucalgary.ca

ABSTRACT
The term "context-free grammar" invariably implies use of a static, unchanging set of grammar rules. When this constraint is removed, dynamic grammars result. In the past, use of dynamic grammars has been relegated to semantics work. However, we show that by defining and interpreting dynamic grammars appropriately, we can create a new grammar-based program execution model. This model unifies disparate areas such as compiler construction, operating systems, data compression, formal languages, interprocess communication, and program representation. Application areas include code compression in embedded systems and the implementation of user-level threads.

KEYWORDS
Compilers, operating systems, data compression, formal languages

1 Introduction

Context-free grammars are primarily used in parsers, which take an input string and attempt to determine if the input is valid according to a set of grammar rules. However, such grammars[1] may be interpreted in at least three other ways [1]. One way to see a grammar is as a generative device, where the grammar rules are applied to generate a sentence; effectively, this is the reverse of what parsers do.

Typically, a parser will use a set of grammar rules that is static – defined at compiler-build time, no grammar rules will be added or deleted as the parser operates. However, it is possible to lift this restriction, resulting in "dynamic grammars."

Dynamic grammars have been the subject of occasional research, dating back to the early 1960s [2]; Christiansen [3] gives a survey of work prior to 1990. However, this work [2, 3, 4, 5, 6, 7] is uniformly concerned with developing a formalism that captures static semantic properties in syntax. For example, upon seeing a variable declaration like

```
int foo;
```

one could dynamically add a grammar rule permitting `foo` to be an identifier. Semantically correct use of `foo` would thus become a syntactic property to be checked by the parser.

To the best of our knowledge, no one else has explored applications of dynamic grammars in a generative sense. We define dynamic grammars and our semantics for them in Section 3. In Sections 4 through 6 we show how dynamic grammars can be used to create a new program execution model, and extend that to a multitasking kernel and the implementation of inter-process communication. We conclude with a discussion of application areas for our execution model.

2 Notation and Definitions

We use standard notation for context-free grammars [8]. Formally, a *context-free grammar* is a quadruple $G = (N, T, R, S)$ where N is a set of *nonterminal* symbols, T is a set of *terminal* symbols ($N \cap T = \emptyset$), R is a set of *grammar rules* of the form $N \times (N \cup T)*$, and S is a distinguished nonterminal called the *start symbol*. When discussing context-free grammars abstractly, we use uppercase letters early in the alphabet to denote nonterminals, lowercase letters early in the alphabet to represent terminals, and uppercase letters late in the alphabet stand for either terminals or nonterminals. The empty string is written as ϵ, w stands for a string of terminal symbols, and α and β stand for strings of terminal and nonterminal symbols $X_1X_2 \ldots X_k, k \geq 0$. Grammar rules are written $A \rightarrow \alpha$.

Beginning with the start symbol, a nonterminal A may be *expanded* by replacing A with α for some grammar rule $A \rightarrow \alpha$. These *derivation steps* may be repeated until only terminal symbols remain, at which point we have a *sentence* in the language described by G. Each string of terminal and nonterminal symbols resulting from a derivation step is called a *sentential form*.

3 Dynamic Grammars

We define our dynamic grammars in a way that gives them some characteristics which we exploit later. Our grammar rules can be partitioned into two types: static and dynamic

[1] Unless otherwise noted, we use "grammar" to mean "context-free grammar" in the remainder of the paper.

$$
\begin{array}{rcl}
(\beta, (N, N_\Delta, T, R, S)) & \vdash_{+A\to\alpha} & (\beta, (N, N_\Delta, T, \{R_1, R_2, \ldots, R_{|R|}, A \to \alpha\}, S)) \\
(wA\beta, (N, N_\Delta, T, R, S)) & \vdash_A & (wA\beta, (N, N_\Delta, T, R, S)), \\
 & & \quad \text{if no rule } A \to \alpha \in R \\
(wA\beta, (N, N_\Delta, T, R, S)) & \vdash_A & (w\alpha\beta, (N, N_\Delta, T, R, S)), \\
 & & \quad \text{if } A \in N, \text{ and } A \to \alpha \text{ is the first rule} \\
 & & \quad \text{in } R \text{ with } A \text{ as the left-hand side} \\
(wA\beta, (N, N_\Delta, T, R, S)) & \vdash_A & (w\alpha\beta, (N, N_\Delta, T, R \setminus \{A \to \alpha\}, S)), \\
 & & \quad \text{if } A \in N_\Delta, \text{ and } A \to \alpha \text{ is the first} \\
 & & \quad \text{rule in } R \text{ with } A \text{ as the left-hand side}
\end{array}
$$

Figure 1. Semantics of transitions between derivation configurations

$$
\begin{array}{rcl}
(S, G_\Delta) & \vdash_S & (aA, (N, N_\Delta, T, \{S \to aA, B \to d\}, S)) \\
 & \vdash_{+A\to bA} & (aA, (N, N_\Delta, T, \{S \to aA, B \to d, A \to bA\}, S)) \\
 & \vdash_{+A\to cB} & (aA, (N, N_\Delta, T, \{S \to aA, B \to d, A \to bA, A \to cB\}, S)) \\
 & \vdash_A & (abA, (N, N_\Delta, T, \{S \to aA, B \to d, A \to cB\}, S)) \\
 & \vdash_A & (abcB, (N, N_\Delta, T, \{S \to aA, B \to d\}, S)) \\
 & \vdash_B & (abcd, (N, N_\Delta, T, \{S \to aA, B \to d\}, S))
\end{array}
$$

Figure 2. Derivation of $abcd$ using a dynamic grammar

rules. A grammar initially only has a set of static grammar rules; as the grammar is used (to derive a terminal string, for instance), other rules may be added dynamically. The two types of rules are treated slightly differently, in that a static grammar rule may never be deleted. In contrast, a dynamic grammar rule is automatically deleted from the grammar when it is used.

To formalize this, a *dynamic context-free grammar* is a quintuple $G_\Delta = (N, N_\Delta, T, R, S)$. N is a set of nonterminal symbols which appear on the left-hand side of static grammar rules, N_Δ is a set of nonterminals which may be used on the left-hand side of dynamic grammar rules ($N \cap N_\Delta = \emptyset$), T is a set of terminal symbols ($N \cap T = N_\Delta \cap T = \emptyset$), and S is the start symbol ($S \in N$). R is an *ordered* set of grammar rules, containing both static grammar rules of the form $N \times (N \cup N_\Delta \cup T)*$ and dynamic grammar rules of the form $N_\Delta \times (N \cup N_\Delta \cup T)*$. We write R_i to refer to the i^{th} rule in R.

When deriving a sentence with a dynamic grammar, is it not sufficient to simply keep track of the evolving sentential form. Both the sentential form and the state of the dynamic grammar may change during a derivation, so changes to both must be noted. A *derivation configuration* for a dynamic grammar G_Δ is a pair $\mathcal{C} = (\beta, G_\Delta)$, where β is a sentential form.

Transitions from one derivation configuration to another, written $\mathcal{C}_i \vdash \mathcal{C}_{i+1}$, may be made in one of two ways. First, the leftmost nonterminal[2] in the sentential form may be expanded ($\mathcal{C}_i \vdash_A \mathcal{C}_{i+1}$). Second, a dynamic grammar rule may be added ($\mathcal{C}_i \vdash_{+A\to\alpha} \mathcal{C}_{i+1}$). The semantics of these operations are given in Figure 1.

For example, let $G_\Delta = (N, N_\Delta, T, R, S)$, where $N = \{S, B\}, N_\Delta = \{A\}, T = \{a, b, c, d\}$, and $R = \{S \to aA, B \to d\}$. The string $abcd$ may be derived as shown in Figure 2.

4 Code as Dynamic Grammar

How can these dynamic grammars be applied? The answer lies in how we interpret the terminal and nonterminal symbols of the grammar.

Compiled code for a program can easily be represented by a static grammar. Compiled code is a static, finite-length sequence of machine instructions. Taking terminal symbols to represent machine instructions, a single grammar rule suffices to describe a sequence of code. Grammars can also be used to describe feasible execution paths in programs, for purposes of program analysis [9]. But in both cases, this is a static description: static code, static analysis.

Instead, we use dynamic grammars to describe a dynamic, executing program. The grammar rules are used to generate the program's code on demand as it is executing. Program execution thus alternates with code generation from grammar rules; every time execution reaches a nonterminal, control reverts back to a "kernel" which expands the nonterminal, then resumes execution.

[2] We only consider leftmost derivations here.

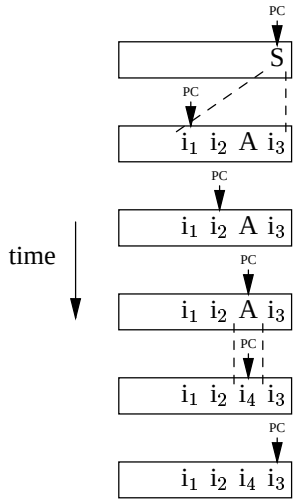

Figure 3. Grammar-based execution model over six time steps. Dashed lines represent the expansion of a nonterminal by the kernel, and "PC" is the program counter

L1: i_1
i_2
goto L2
...
L2: i_3

$\Rightarrow$

L1 $\rightarrow$ i_1 i_2 L2
L2 $\rightarrow$ i_3

(a) Goto

L1: i_1
i_2
call foo
i_3
...
foo: i_4

$\Rightarrow$

L1 $\rightarrow$ i_1 i_2 foo i_3
foo $\rightarrow$ i_4

(b) Function call

Figure 4. Translation of unconditional control flow into grammar rules

Figure 3 illustrates this new execution model, for the grammar:

$$\begin{array}{lcl} S & \rightarrow & i_1\ i_2\ A\ i_3 \\ A & \rightarrow & i_4 \end{array}$$

Here, i_1, i_2, i_3, and i_4 are instructions, and S is the start symbol. At each time step, the box represents the current state of the program code in memory; we call this state the *sentential code form*, because it corresponds to a sentential form of the grammar. The initial execution state simply has the start symbol as the sentential code form, with the program counter pointing to the start symbol. As the start symbol is a nonterminal, this causes the kernel to expand the nonterminal and adjust the program counter accordingly. In this particular example, the kernel performs two nonterminal expansions, and no dynamic grammar rules are used.

L1: i_1
if cond goto L2
else goto L3
...
L2: i_2
...
L3: i_3

$\Downarrow$

L1 $\rightarrow$ i_1
[cond ? gen L1′ $\rightarrow$ L2]
[¬cond ? gen L1′ $\rightarrow$ L3]
L1′
L2 $\rightarrow$ i_2
L3 $\rightarrow$ i_3

Figure 5. Translation of a conditional goto into grammar rules. Machine instruction details are enclosed in brackets

One interesting aspect of our grammar-based execution model is that programs do not need any explicit control transfer instructions; there is sufficient mechanism in the grammar model to handle control flow changes.

Let us assume that user programs are compiled to a typical assembly code. The assembly code can be converted into a grammar by creating a separate grammar rule for each basic block in the code. There are no user-visible changes because we convert from the assembly level; users write their programs as they normally would, in any language.

Given this grammar encoding, a goto statement in the assembly code is simply replaced by a nonterminal naming the basic block that was the target of the goto. This is illustrated in Figure 4a. When execution reaches the end of $L1$ and the program counter points to $L2$, the kernel will expand $L2$, yielding the net effect of the original goto.

Function calls are dealt with in a similar manner, as Figure 4b shows. A function call in the assembly code is replaced by the nonterminal corresponding to the start of the callee's code. Essentially, this results in procedure integration [10] of the callee into the caller's code.

Unconditional gotos and function calls are static, in that their targets are unchanging. What about conditional gotos? The answer lies in dynamic generation of grammar rules, shown in Figure 5. Here, the nonterminal $L1'$ is initially without a definition. Execution of $L1$'s code results in the generation of either $L1' \rightarrow L2$ or $L1' \rightarrow L3$, depending on the value of the conditional. To ensure that the correct rule is generated, dynamic rule-generation instructions may be *guarded* – an instruction is only executed if its guard expression is true [11]. This is only a partial solution, because the program may need to take a different path the next time the conditional is evaluated. Conditional gotos function as expected, however, because our semantics for dynamically-generated grammar rules cause a dynamically-generated rule to be deleted immediately after

being used in a production by the kernel. In other words, a dynamically-generated rule does not remain in the dynamic grammar to incorrectly influence future program execution.

Let us now formalize the dynamic grammar-based execution model. At any point during execution, the execution state may be described using a triple $(pc, \mathcal{C}, Q)$. $\mathcal{C}$ is a derivation configuration for a dynamic grammar describing the program ($\mathcal{C}$'s sentential form β is therefore a sentential code form), pc is used to index into the sentential code form ($1 \leq pc \leq |\beta| + 1$), and Q is a code processor state.

The code processor is what actually executes individual instructions, such as the assembly code instructions we used earlier. The instructions it supports are irrelevant, so long as two constraints hold:

1. Instruction execution must be sequential; no control flow instructions are permitted. Control flow changes may be handled using the dynamic grammar mechanisms described above.

2. A guarded instruction must be provided to dynamically generate grammar rules.

These constraints can be easily met if the code processor is a virtual machine implemented in software. For a hardware code processor – a CPU – some creativity may be required. In the ARM architecture [12], for instance, the second constraint may be met using a conditionally-executed branch instruction.

The state of the code processor is abstracted away by Q, upon which two operations are defined. First, we need to know the new code processor state following execution of an instruction i, written $Q' = execute_i Q$. Second, we must be able to evaluate a side-effect-free predicate b in Q, and receive a boolean value in return. This second operation is denoted $query_b Q$. The initial code processor state is called Q_0.

Returning to the execution model, the state transitions are described in Figure 6. Indexing the sentential code form using the program counter is denoted $\beta[pc]$ and is defined only when $1 \leq pc \leq |\beta|$. The set T_{gen}, $T_{gen} \subseteq T$, consists of terminal symbols that correspond to guarded dynamic rule-generation instructions of the form $b?A \rightarrow \alpha$. These deterministic state transitions are applied repeatedly, starting from the initial execution state $(1, (S, G_\Delta), Q_0)$ until an execution state is reached where $pc > |\beta|$.

$$\boxed{\begin{array}{lll} & pc & \mathcal{C} & Q \\ \Longrightarrow & pc & \mathcal{C}' & Q \\ & \text{where } \beta[pc] \in (N \cup N_\Delta) \end{array}}, \mathcal{C} \vdash_{\beta[pc]} \mathcal{C}'$$

$$\boxed{\begin{array}{lll} & pc & \mathcal{C} & Q \\ \Longrightarrow & pc+1 & \mathcal{C} & execute_{\beta[pc]} Q \\ & \text{where } \beta[pc] \in (T \setminus T_{gen}) \end{array}}$$

$$\boxed{\begin{array}{lll} & pc & \mathcal{C} & Q \\ \Longrightarrow & pc+1 & \mathcal{C}' & Q \\ & \text{where } \beta[pc] \in T_{gen} \text{ and } query_b Q = true \end{array}}, \quad \mathcal{C} \vdash_{+A \rightarrow \alpha} \mathcal{C}'$$

$$\boxed{\begin{array}{lll} & pc & \mathcal{C} & Q \\ \Longrightarrow & pc+1 & \mathcal{C} & Q \\ & \text{where } \beta[pc] \in T_{gen} \text{ and } query_b Q = false \end{array}}$$

Figure 6. State transitions of dynamic grammar-based execution model

5 A Multitasking Kernel

When control flow is encoded as described in the last section, the kernel is frequently invoked to expand nonterminals. Also, programs cannot loop without the kernel being called, because there are no explicit control flow instructions. The kernel is in an ideal position to implement nonpreemptive context switching.

Nonpreemptive, or cooperative, multitasking has been used previously in operating systems, most notably in early versions of MacOS [13, 14] and Windows [15, 16]. In these systems, programs are event-driven and assumed to frequently call an API routine to get the next event; the operating system relies on this to regain control and context switch to another process. Any violation of this assumption by a program – a long-running computation, a misbehaving process – and the system as a whole becomes unresponsive. In contrast, our nonpreemptive multitasking does not require programs to be written in a particular fashion and cannot lose control of the system. Successful nonpreemptive multitasking follows from the underlying program representation, not the program behavior.

In the kernel, each process is represented by an execution state as described in the previous section. Part of the execution state is the code processor's state which, in concrete terms, would typically consist of a register set, a stack, and a heap. The set of processes is assumed to be fixed – a feasible constraint for many embedded systems. Every time the kernel is invoked to expand a nonterminal, it suspends the current process, then finds and resumes the next unblocked process in round-robin[3] fashion. But how can a process block in the first place?

6 Inter-Process Communication

Blocking and inter-process communication in our grammar-based execution model hinge on the concept of "useless" nonterminals, specifically undefined nonterminals. Normally the appearance of an undefined nonterminal in a grammar rule would signal a potential error, and the grammar rule would be deleted [17]. In a dynamic grammar, an undefined nonterminal makes it im-

[3]Of course, variations on the scheduling algorithm or context switching policy are possible.

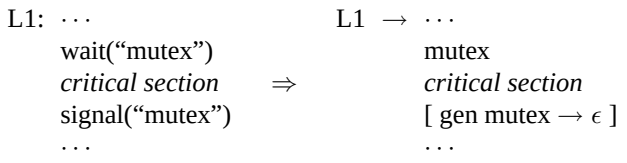

Figure 7. Semaphore translation

possible for the kernel to expand the sentential code form further; this is formalized by the semantics in Figure 1. We use this idea as the basis of blocking – a process whose program counter points to an undefined nonterminal is blocked. That semantics, plus the fact that nonterminal expansion is an atomic kernel operation, allows for the construction of inter-process communication mechanisms.

Semaphores are exposed to the user via `signal` and `wait` function calls at the source language level. The compiler maps these into a dynamic rule generation and an undefined nonterminal, respectively, as shown in Figure 7. In the user's code, "mutex" is used as the semaphore's name; this becomes an undefined nonterminal in the translated code. A minor extension to program execution semantics is required here, so that one process may add grammar rules to another process' dynamic grammar. (A simple way to implement this is for the kernel to maintain all process' grammar rules globally, and prefix each nonterminal's name with the name of the process: nonterminal A in process *P1* would become *P1*_A.)

Note that our dynamic grammars apply dynamically-generated rules using a FIFO scheme, so multiple signals on a single semaphore are handled properly. This ordering property is also useful for the second type of inter-process communication we support, messages.

Messages are used for passing values between processes. The user simply sees two operations at the source language level, `send` and `recv`. Their translation is given in Figure 8, where the integer value 123 is being sent. The receiver waits on the "queue" nonterminal for a message, after which it saves the message into `x`. The compiler translates `recv` as a function returning a value, where function return values are placed in register `$rv`. The sender sends its message by generating a rule which causes "queue" to be replaced by a single instruction that loads the appropriate value into `$rv`.

Essentially, message passing is implemented by having one process dynamically modify another process' code. What we have constructed is a formal framework for self-modifying code as well as run-time code generation [18]. Our grammar-based execution model allows a process to declare where it can be modified, and provides a mechanism for doing so: grammar rule generation.

We have written a proof-of-concept implementation which employs a virtual machine as a code processor. Currently, we are looking at ways to implement grammar-based execution efficiently on the ARM processor.

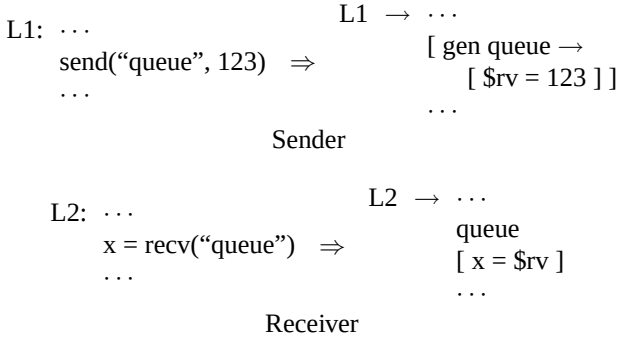

Figure 8. Message translation

7 Applications

Our grammar-based execution model allows for code decompression as part of its basic operation, an attractive feature for memory-limited embedded systems. Employing a grammar-based scheme for data compression, repetitive code sequences become new grammar rules, with the original locations of these sequences compressed to a single nonterminal [19]. The kernel decompresses code compressed this way automatically during program execution.

Software-based code decompression has been explored, for example, in [20, 21, 22]. Our approach is loosely related to Liao et al., who extract common code sequences and invoke them via function calls at runtime [23]. Our current contribution is not a new compression method, but a natural way to incorporate decompression into program execution.

Grammar-based execution may also be applicable in the implementation of user-level thread packages, or any situation where multitasking is required, but preemptive multitasking is not feasible.

8 Conclusion

We have explored a practical application of dynamic context-free grammars, where the grammars are used to generate sentences rather than parse them. By interpreting the grammar's terminal symbols as instructions, we arrive at a grammar-based execution model for programs. We are thus able to unify ideas in compiler construction, operating systems, data compression, formal languages, inter-process communication, and program representation.

Other unification efforts have been made. Symbolics Lisp machines permeated all software with a single programming language, and allowed a user to replace any component, including ones in the operating system [24]. The EZ system involved a programming language which subsumed operating system functionality into language abstractions mapping, for example, files into strings [25]. Jones [26] suggests ways to exploit concepts common to both languages and operating systems, and also advocates that user programs and the operating system be written in

the same language.

However, our novel grammar-based execution model changes only the low-level program representation. We are thereby able to unify many areas without impinging on the user's choice of programming language. The user never sees the grammar mechanisms and is free to write programs for our system in any programming language.

9 Acknowledgments

Shannon Jaeger made many helpful comments on this paper. This work was supported in part by the National Science and Engineering and Research Council of Canada.

References

[1] R. Leermakers, *The Functional Treatment of Parsing* (Boston: Kluwer Academic, 1993).

[2] A. C. di Forino, Some remarks on the syntax of symbolic programming languages, *Communications of the ACM*, 6(8), 1963, 456–460.

[3] H. Christiansen, A survey of adaptable grammars, *ACM SIGPLAN Notices*, 25(11), 1990, 35–44.

[4] P. Boullier, Dynamic grammars & semantic analysis, Technical Report 2322, INRIA, 1994.

[5] B. Burshteyn, On the modification of the formal grammar at parse time, *ACM SIGPLAN Notices*, 25(5), 1990, 117–123.

[6] S. Cabasino, P. S. Paolucci, and G. M. Todesco, Dynamic parsers and evolving grammars, *ACM SIGPLAN Notices*, 27(11), 1992, 39–48.

[7] K. V. Hanford and C. B. Jones, Dynamic syntax: A concept for the definition of the syntax of programming languages, *Annual Review in Automatic Programming*, volume 7, 1974, 115–142.

[8] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools* (Reading, Mass.: Addison-Wesley, 1988).

[9] T. Reps. Program analysis via graph reachability. *Information and Software Technology*, 40(11–12), 1998, 701–726.

[10] S. S. Muchnick, *Advanced Compiler Design and Implementation* (San Francisco: Morgan Kaufmann, 1997).

[11] P. Y. T. Hsu and E. S. Davidson. Highly concurrent scalar processing. *Proc. 13th Annual Int'l Symposium on Computer Architecture*, Tokyo, Japan, 1986, 386–395.

[12] D. Seal, editor, *ARM Architecture Reference Manual*, second edition (Reading, Mass.: Addison-Wesley, 2000).

[13] Apple Computer, Inc., *Technical Introduction to the Macintosh Family*, second edition, (Reading, Mass.: Addison-Wesley, 1992).

[14] G. Little and T. Swihart, *Programming for System 7* (Reading, Mass.: Addison-Wesley, 1991).

[15] N. Baran, Operating systems now and beyond, *BYTE*, 16(11), 1991, 93–98.

[16] R. C. Kennedy, The elegant kludge, *BYTE*, 20(8), 1995, 54–60.

[17] D. Grune and C. J. H. Jacobs, *Parsing Techniques: A Practical Guide* (New York: Ellis Horwood, 1990).

[18] D. Keppel, S. J. Eggers, and R. R. Henry, A case for runtime code generation, Technical Report 91-11-04, University of Washington Department of Computer Science and Engineering, 1991.

[19] C. G. Neville-Manning and I. H. Witten, Compression and explanation using hierarchical grammars, *Computer Journal*, 40(2/3), 1997, 102–116.

[20] K. D. Cooper and N. McIntosh, Enhanced code compression for embedded RISC processors, *Proc. ACM SIGPLAN '99 Conference on Programming Language Design and Implementation*, Atlanta, USA, 1999, 139–149.

[21] W. S. Evans and C. W. Fraser, Bytecode compression via profiled grammar rewriting, *Proc. ACM SIGPLAN '01 Conference on Programming Language Design and Implementation*, Snowbird, USA, 148–155.

[22] C. Lefurgy, E. Piccininni, and T. Mudge, Reducing code size with run-time decompression, *Proc. 6th Int'l Symposium on High-Performance Computer Architecture*, Toulouse, France, 2000, 218–227.

[23] 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), 1999, 12–38.

[24] J. H. Walker, D. A. Moon, D. L. Weinreb, and M. McMahon, The Symbolics Genera programming environment, *IEEE Software*, 4(6), 1987, 36–45.

[25] C. W. Fraser and D. R. Hanson, Integrating operating systems and languages, Technical Report TR 84-2, Department of Computer Science, University of Arizona, 1984.

[26] A. K. Jones, The narrowing gap between language systems and operating systems, *Information Processing 77*, Toronto, Canada, 1977, 869–873.