A Declarative Approach to Run-Time Code Generation

Mark Leone Peter Lee

School of Computer Science
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213-3891

{mleone,petel}@cs.cmu.edu

1 Introduction

Run-time code generation promises to improve the performance and reliability of current and future systems. Optimizations performed at run time make use of values and invariants that cannot be exploited at compile time, yielding code that is often superior to statically optimal code. Furthermore, run-time code generation encourages better structuring of systems: in our experience, the performance of special-purpose code can, in some cases, be matched or exceeded by automatically customized general-purpose code. Run-time code generation also promotes the use of advanced languages because it amortizes the cost of dynamic safety checks and optimizes across abstraction boundaries.

Most previous approaches to run-time code generation have been *imperative*, relying on the programmer to:

- specify the code to be optimized at run time, usually as a "template" of machine code or as code that builds an abstract syntax tree,
- substitute run-time values into code and combine code fragments,
- perform certain optimizations, such as loop unrolling and strength reduction,
- insure that dynamically constructed code fragments are well formed,
- invoke a compiler or template instantiator and dynamically link to the resulting code,
- reuse run-time-generated code and reclaim code space, and
- insure correctness by invalidating or updating code when the values or invariants used to optimize it change.

Each of these duties is laborious and error prone. Various ad-hoc techniques have been suggested to ameliorate this situation. For example, libraries for dynamically compiling and linking code now exist [7], and various notations and primitives have been suggested to simplify specifying and combining code fragments [5]. However, we believe that an imperative approach to run-time code generation is clumsy and inherently unsafe.

We advocate a *declarative* approach to run-time code generation: the programmer should express algorithms in a high-level language that permits the *compiler* to discover and safely exploit opportunities for dynamic optimization. This may appear to be an impossibly lofty goal; on the contrary, we have early evidence from experiments with a prototype compiler that automatic run-time code generation can be practical. Our compiler, called Fabius, compiles ordinary programs written in a subset of Standard ML into code that generates and executes native code at run time.

This research was sponsored in part by the Advanced Research Projects Agency CSTO under the title "The Fox Project: Advanced Languages for Systems Software", ARPA Order No. C533, issued by ESC/ENS under Contract No. F19628-95-C-0050, and in part by the National Science Foundation under PYI grant #CCR-9057567.

The key benefits of a declarative approach to run-time code generation are:

- Safety: the programmer does not write explicit code for generating code, but instead submits an ordinary program to the compiler.
- Portability: the compiler encapsulates all of the details of run-time code generation.
- Principled design: the compiler exploits previous work on partial evaluation of declarative languages.
- Speed: partial evaluation techniques are used to produce specialized run-time code generators that do not manipulate intermediate code at run time.

We now give a brief overview of how procedures written in ML can be compiled into extremely fast run-time code generators. Then, we present an example that demonstrates the effectiveness of our approach: an interpreter for the BSD packet filter language.

2 Specialized Run-Time Code Generation

The ML programming language encourages a more declarative style of programming. This facilitates the safe and easy exploitation of staged computations via run-time code generation. For example, data structures in ML programs are almost always immutable, so values may be freely copied by an optimizer. This is an important safety criteria, since most run-time optimizations are value specific: creating code optimized to a particular value can cause a coherency problem if the value can be modified later. Furthermore, much of the benefit of run-time code generation arises from *staging* computations so that values computed in "early" stages can be used to optimize the code for "later" stages. Such staging is simplified by the extensive use of immutable values: many expressions in a program will have no side effects, which makes it easier to reorder them.[1]

ML also allows the programmer to express staged computations in a natural way through the use of higher-order functions. When a function $f : \tau_1 \rightarrow (\tau_2 \rightarrow \tau_3)$ is applied to an argument $x : \tau_1$, it returns a function of type $\tau_2 \rightarrow \tau_3$. Hence, it is natural to regard x as an "early" value and create optimized code for the functional result of $f(x)$ rather than simply building a closure.

In essence, run-time code generation is a form of partial evaluation [9], a fact that was first noted by Massalin [14]. A partial evaluator is a function (called *mix* for historical reasons) that takes the code of a function f and an argument x, and returns optimized code for the result of applying f to x:

$$mix(f, x) \;=\; f_x$$

The result of this optimization is code for a *specialized* function, $f_x : \tau_2 \rightarrow \tau_3$, that computes the same function as $f(x)$, but which does so without repeating computations that depend only upon x. In a similar way, partial evaluation can be used to reduce the cost of run-time code generation: the text of the functions *mix* and f are known at compile time, so we can specialize *mix* to f:

$$mix(mix, f) \;=\; mix_f$$

The result is the code for a specialized code generator, $mix_f : \tau_1 \rightarrow (\tau_2 \rightarrow \tau_3)$, that can be used to compute f_x without the overhead of processing the text of f. In essence, the intermediate

[1] Of course, with sufficiently advanced analyses it may be possible to discover similar properties about programs written in imperative languages like C. But past experience with program transformations like partial evaluation and automatic parallelization have shown that such analyses are extremely difficult for real-world programs.

representation of f is "pre-compiled" away, so that it is no longer needed when the argument x is supplied.

These ideas, which are taken from the field of partial evaluation, are used by Fabius to create specialized run-time code generators that do not manipulate any intermediate representation of code at run time. Rather than describing precisely how this is done (which we have done elsewhere [13, 12]), we will instead present an intuitive example to illustrate our techniques: an interpreter for a simple packet filter language.

3 An Example: Packet Filtering

A packet filter is a procedure invoked by an operating system kernel to select network packets for delivery to a user-level process. To avoid the overhead of context switching on every packet, a packet filter must be kernel resident. But this has a distinct disadvantage: it can be difficult for user-level processes to specify precisely the types of packets they wish to receive, since packet selection criteria can be quite complicated. Many useless packets may be delivered as a result, with a consequent degradation of performance.

A commonly adopted solution to this problem is to parameterize a packet filter by a selection predicate that is dynamically constructed by a user-level process [18, 17]. A selection predicate is expressed in the abstract syntax of a "safe" or easily verified programming language, so that it can be trusted by the kernel. But this approach has significant overhead: the selection predicate is re-interpreted every time a packet is received.

Run-time code generation can eliminate the overhead of interpretation by compiling a selection predicate into trusted native code. More generally, run-time code generation can allow a kernel to efficiently execute "agents" supplied by user-level processes while avoiding context switches. This idea has also been investigated by others [6, 1].

3.1 The Interpreter

To demonstrate how this is possible, we consider an interpreter for the BSD packet filter language [17]. The interpreter shown in Figure 1 is an ordinary ML function, called `eval`, that is parameterized by the filter program, a network packet, and variables that encode the machine state. The interpreter is curried, since it will be applied to the same filter program many times.

Fabius compiles `eval` into a code generator that *performs* all of the operations involving the filter program and *generates code* for the operations involving the packet data and machine state. For example, the instruction decoding and dispatch operations in `eval` are compiled into the following MIPS code:[2]

```
eval:  addi   $pc, $pc, 2
       lw     $length, -4($filter)    ; Vector is tagged with a length field
       slt    $tmp, $pc, $length      ; pc >= length of filter?
       beq    $tmp, $0, L1
       sll    $i, $pc, 2              ; Shift to get byte offset of instruction
       addu   $p, $filter, $i         ; Compute address of instruction
       lw     $instr, ($p)            ; instr = filter sub pc
       srl    $opcode, $instr, 16     ; opcode = instr >> 16
```

[2]To clarify this and other examples, we omit delay slots and some range-checking code for subscript operations, and we use untagged integers and pointers.

```
(* The BSD packet filter language is comprised of RISC-like instructions for a simple
   abstract machine with an accumulator, an index register, and a small scratch memory.
   The abstract machine is made "safe" by confining memory references to the packet data
   and scratch memory and by forbidding backwards jumps. Instructions are encoded as pairs
   of 32-bit words. The first word specifies a 16-bit opcode and, optionally, a pair of
   8-bit branch offsets. The second word specifies an immediate value that can be used for
   ALU operations and memory indexing. *)

fun eval (filter, pc) (a, x, mem, pkt) =
 let val pc = pc + 2
 in
    if pc >= length filter then ~1 else
    let val instr = filter sub pc
        val opcode = instr >> 16
    in
        (* Add immediate to accumulator. *)
        if opcode = ADD_K then
           let val k = filter sub (pc+1)
           in
              eval (filter, pc) (a + k, x, mem, pkt)
           end
        ...
        (* Jump if accumulator equals immediate. *)
        else if opcode = JEQ_K then
            if a = filter sub (pc+1) then
                eval (filter, pc + ((instr >> 8) andb 255)) (a, x, mem, pkt)
            else eval (filter, pc + (instr andb 255)) (a, x, mem, pkt)
        ...
    end
 end
```

Figure 1: Packet filter interpreter

```
        li      $tmp, ADD_K
        beq     $opcode, $tmp, L2
        ...
        li      $tmp, JEQ_K
        beq     $opcode, $tmp, L3
        ...
```

So far the code looks rather ordinary. But consider the branch of the interpreter corresponding to the ADD_K instruction, which adds an immediate value (specified by the filter program) to the accumulator:

```
L2:     addi    $i, $pc, 1              ; Immediate is at pc+1
        sll     $i, $i, 2               ; Shift to get byte offset
        addu    $p, $filter, $i         ; Compute address of immediate
        lw      $k, ($p)                ; k = filter sub (pc+1)
        emit    addi $a, $a, $k         ; Add immediate (in $k) to accumulator
        j       eval                    ; Recursive tail call
```

This code performs the computations involving the filter program and the `pc`, and it *emits* code for an operation involving the accumulator, since the accumulator is one of the "late" arguments to `eval`.

The `emit` pseudo-instruction is actually a short sequence of instructions that construct the native encoding of the instruction to be emitted and write it into a dynamic code segment. For example, the `addi` instruction above (which uses the value in `$k` as an immediate) is emitted as follows:[3]

```
        lui     $tmp, 0x2063            ; Load ADDI opcode into upper half
        or      $tmp, $tmp, $k          ; Load immediate into lower half
        sw      $tmp, ($cp)             ; Store instruction
        addiu   $cp, $cp, 4             ; Advance code pointer
```

In effect, FABIUS compiles `eval` into a run-time code generator that emits optimized code directly, without manipulating any intermediate representation at run time. This makes the entire process of dynamic code generation extremely lightweight, requiring an average of just six cycles per instruction generated. Unfortunately, there are quite a few complications and more interesting opportunities for optimization that are not shown here; some are described elsewhere [12].

3.2 The Run-Time-Generated Code

Despite the simplicity of the optimizations applied, `eval` generates high quality code. For example, consider the following filter program (written in BPF pseudo-code), which selects packets whose Ethernet type field specifies an IP packet:[4]

```
        LD   4                  ; Load 5th packet word into accumulator.
        RSH  16                 ; Shift, yielding type field.
        JEQ  ETH_IP, L1         ; Is this an IP packet?
        RET  1                  ; If so, accept.
   L1:  RET  0
```

[3] We assume here that `$k` can be used as a 16-bit immediate value. In fact, FABIUS uses additional code to check this condition and emit a more general sequence of instructions if necessary.

[4] For simplicity of presentation, this packet filter and the code generated for it do not use byte swapping.

The specialized code generator for `eval` produces the following MIPS code for this packet filter at run time:

```
        li   $offset, 4                  ; Offset in words
        lw   $length, -4($p)             ; Get packet size (in words)
        slt  $tmp, $offset, $length      ; Compare load offset to packet size.
        beq  $tmp, $0, L1
        lw   $a, 16($pkt)                ; Load 5th packet word into accumulator.
        srl  $a, $a, 16                  ; Shift, yielding type field.
        li   $tmp, ETH_IP
        beq  $a, $tmp, L2                ; Is this an IP packet?
        li   $result, 0                  ; Reject if not.
        j    L3
L2:     li   $result, 1                  ; Else accept.
L3:     j    L4
L1:     li  $result, -1                  ; Return -1 if indexing error.
L4:     jr   $ra
```

This code is not quite optimal because the run-time code generator has failed to eliminate two jumps whose targets are jumps. But it does demonstrate several interesting run-time optimizations:

- The most beneficial optimization is a kind of run-time loop invariant removal: all operations involving the packet filter and the program counter, such as instruction fetching and decoding, have been performed by the code generator.
- Run-time "constant" propagation has embedded values from the filter program (such as the load offset, the shift amount, and the `ETH_IP` constant) as immediates in the run-time generated code.
- Run-time "constant" folding has also occurred: because the load offset is a run-time "constant," the code generator has folded the scaling implicit in the subscripting operation. The resulting byte offset is used in a load-immediate instruction.
- Run-time inlining has eliminated all of the tail-recursive calls in `eval`. These jumps are performed by the code generator, not the generated code.

Larger packet filter programs show similar improvements. In the next section, we show the performance of the code generated by Fabius using a more complicated packet filter.

4 Results

Figure 2 compares the overall execution times of the Fabius packet filter implementation (including time spent generating code) to the BPF implementation in C [16], using a packet filter that selects non-fragmentary TCP/IP packets destined for a Telnet port. To reliably compare execution times, we obtained five sample packet traces by eavesdropping on a busy CMU network, and we averaged execution times over these traces as a precaution against abnormal packet mixes. We modified the BPF interpreter to read packets from these trace files, compiled it with `-O2` optimization, and executed it in user mode. The timings reported here exclude the time required to read packets from the trace files.

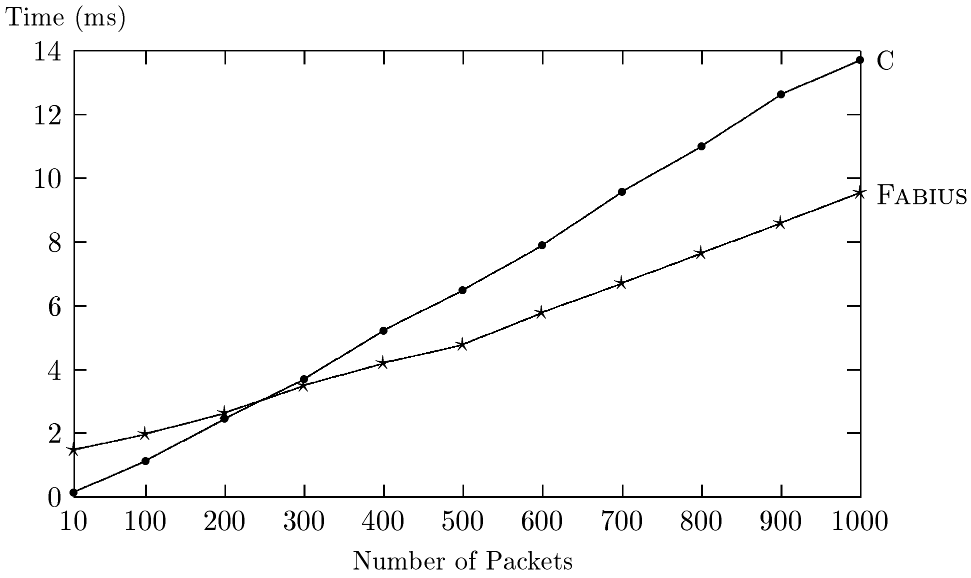

Figure 2: Run-time code generation for a packet filter

As the figure demonstrates, the time spent generating code at run time was quickly repaid: the FABIUS implementation broke even with BPF after approximately 250 packets. After only 1000 packets, execution time was reduced by 30.3%; as the number of packets increases, the reduction in execution time approaches 40%. The time spent generating code was brief, totalling 1.3 ms. Instrumentation revealed that the run-time code generator executed an average of only 5.6 instructions per instruction generated. The run-time generated code required an average of 8.3 μs to process a packet, whereas BPF averaged 13.7 μs per packet.

Space usage was insignificant. The abstract syntax of the packet filter program occupied 34 words in both implementations, and the FABIUS implementation generated 85 instructions at run time. The run-time code generator performed a small amount of heap allocation (43 words), but none was required by the run-time generated code. Stack usage was also insignificant, since tail-recursive calls were compiled into jumps.

We have also experimented with a variety of other programs, ranging from numerical to symbolic programs, with similar (and in some cases, much better) results [12]. As with the packet-filter example, the code submitted to FABIUS is an ordinary ML program, without any explicit mention of run-time code generation beyond the currying of function arguments to identify staged computations.

5 Related Work

Run-time code generation has a long history; for example, Ken Thompson implemented a run-time compiler for regular expressions three decades ago [22]. Keppel and colleagues have recounted much of this history [10], and have also investigated the tradeoff between the cost of run-time optimization and the benefit obtained [11].

Recent work on run-time code generation has focused primarily on its use in improving the performance of operating systems. [15, 1, 8, 21]. A few other researchers have focused on using run-time code generation as a more general-purpose optimization technique. The work most closely related to ours is that of Consel and Noël [2], who are implementing a general-purpose run-time specializer for C. Like us, they advocate a declarative approach to run-time code generation, where the compiler automatically infers where and how to specialize code at run time. However, automatic specialization of imperative languages like C is less profitable than it is for languages like ML because unrestricted aliasing and mutable data structures require program analyses to be overly conservative, which limits opportunities for optimization.

This limitation can be bypassed by requiring the programmer to specify manually where and how to specialize code at run time. This approach has been adopted by Engler and colleagues, who have implemented a compiler for an extension of C with a LISP-like backquote operator that simplifies the construction of code fragments to be compiled at run time [5]. The programmer writes code that manually combines code fragments and applies certain optimizations, such as loop unrolling. An early compiler for this language was based on DCG [7], a dynamic compiler derived from the back end of the `lcc` compiler. Because DCG processes a general-purpose intermediate representation at run time, it imposes a significant overhead: approximately 350 instructions are executed per instruction generated at run time. A more recent implementation (described in these proceedings [20]) is based on VCODE, a set of portable primitives for emitting native machine code [4]. Backquoted code is compiled into C code that invokes VCODE macros to emit code directly, without processing any intermediate representation at run time.

In a sense, the VCODE implementation of ‘C represents a rediscovery of the partial evaluation techniques that we developed to reduce the overhead of run-time code generation. A backquoted expression is essentially an expression in a simplistic *two-level language* [19]. Two-level languages are the foundation of offline partial evaluation, which we discussed in Section 2. As we demonstrated, partial evaluation allows a compiler to create specialized code generators that manipulate no intermediate representation of code at run time. The ‘C compiler achieves precisely this, but in a more ad-hoc manner.

Since it allows a more declarative specification of run-time optimization, a compiler based on a two-level intermediate representation has several advantages over a compiler for a language with backquote:

- Less programmer effort: ordinary programs can be compiled into a two-level intermediate representation. In effect, an analysis automatically determines where to insert backquote and comma operators.[5]
- Greater generality: backquoted expressions have limited expressiveness. The comma operator can only be used to substitute values and code fragments into a piece of code. A two-level intermediate representation can be transformed [12] or annotated with *actions* [2] to allow a much wider range of optimizations to be expressed.
- Improved safety: although backquoted expressions in ‘C are statically type checked, they are manually combined by the programmer, which can be error prone.
- Principled design: two-level languages have well-understood semantics, and previous work on partial evaluation has carefully considered how specialization affects the behavior of programs.

The use of backquote does have some advantages: since it is used explicitly by the programmer, it provides fine-grained control over the use of run-time optimization, which may lead to greater flexibility and better predictability of program behavior.

[5]The comma operator in LISP is used to substitute values into backquoted expressions. In ‘C this operation is expressed by two distinct operators (`@` and `$`), depending on whether the value being substituted is code or a scalar.

6 Future Directions

We have implemented an approach to run-time code generation that is both principled and, for some realistic examples, practical. The use of ML allows the compiler to perform run-time optimizations with little effort on the part of the programmer. It also facilitates a substantial use of partial evaluation techniques to optimize the optimization process. Further experimentation will be required to fully evaluate the progress that has been made thus far, and we have identified numerous areas for further research. We are currently extending the prototype FABIUS compiler to support a richer source language, including mutable data structures and higher-order functions. We also plan to investigate the feasibility of run-time register assignment, scheduling, and other run-time optimizations.

Our experience with benchmark programs has led us to the conclusion that this technology is usable but not yet foolproof. The use of a high-level programming language greatly reduces the programmer effort required to make use of run-time code generation, but it can be difficult for the programmer to predict a program's behavior. Run-time code generators are memoized to reuse code whenever possible, but this can be expensive and is sometimes ineffective. Also, the heuristic we employ to control run-time inlining occasionally leads to over-specialization or under-specialization. Recent advances in type theory have suggested a mechanism for providing better feedback to programmers [3].

Ultimately, the feasibility of our approach to run-time code generation will have to be demonstrated on larger, more realistic programs. It is encouraging to see that a prototype such as FABIUS can already achieve good results.

References

[1] Brian Bershad, Stefan Savage, Przemyslaw Pardyak, Emin Gun Sirer, David Becker, Marc Fiuczynski, Craig Chambers, and Susan Eggers. Extensibility, safety and performance in the SPIN operating system. In *Proceedings of the 15th ACM Symposium on Operating System Principles*, pages 267–284, December 1995.

[2] Charles Consel and François Noël. A general approach to run-time specialization and its application to C. In *Conference Record of POPL '96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 145–156, January 1996.

[3] Rowan Davies and Frank Pfenning. A modal analysis of staged computation. In *Conference Record of POPL '96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 258–270, January 1996.

[4] Dawson R. Engler. VCODE: A retargetable, extensible, very fast dynamic code generation system. In *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*, May 1996. To appear.

[5] Dawson R. Engler, Wilson C. Hsieh, and M. Frans Kaashoek. 'C: A language for high-level, efficient, and machine-independent dynamic code generation. In *Conference Record of POPL '96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 131–144, January 1996.

[6] Dawson R. Engler, M. Frans Kaashoek, and James W. O'Toole Jr. The operating system kernel as a secure programmable machine. *Operating Systems Review*, 29(1):78–82, January 1995.

[7] Dawson R. Engler and Todd A. Proebsting. DCG: An efficient, retargetable dynamic code generation system. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VI)*, pages 263–272. ACM Press, October 1994.

[8] Dawson R. Engler, Deborah Wallach, and M. Frans Kaashoek. Efficient, safe, application-specific message processing. Technical Memorandum MIT/LCS/TM533, MIT Laboratory for Computer Science, March 1995.

[9] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, 1993.

[10] David Keppel, Susan J. Eggers, and Robert R. Henry. A case for runtime code generation. Technical Report 91-11-04, Department of Computer Science and Engineering, University of Washington, November 1991.

[11] David Keppel, Susan J. Eggers, and Robert R. Henry. Evaluating runtime-compiled value-specific optimizations. Technical Report 93-11-02, Department of Computer Science and Engineering, University of Washington, November 1993.

[12] Peter Lee and Mark Leone. Optimizing ML with run-time code generation. In *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*, May 1996. To appear. A preliminary version is available as Technical Report CMU–CS–95–205, School of Computer Science, Carnegie Mellon University, December 1995.

[13] Mark Leone and Peter Lee. Lightweight run-time code generation. In *PEPM 94 Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 97–106. Technical Report 94/9, Department of Computer Science, University of Melbourne, June 1994.

[14] Henry Massalin. *Synthesis: An Efficient Implementation of Fundamental Operating System Services.* PhD thesis, Department of Computer Science, Columbia University, 1992.

[15] Henry Massalin and Calton Pu. Threads and input/output in the Synthesis kernel. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pages 191–201, 1989.

[16] Steven McCanne. The Berkeley Packet Filter man page. BPF distribution available at `ftp://ftp.ee.lbl.gov`.

[17] Steven McCanne and Van Jacobson. The BSD packet filter: A new architecture for user-level packet capture. In *Proceedings of the Winter 1993 USENIX Conference*, pages 259–269. USENIX Association, January 1993.

[18] Jeffrey C. Mogul, Richard F. Rashid, and Michael J. Accetta. The packet filter: An efficient mechanism for user-level network code. In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, pages 39–51. ACM Press, November 1987. An updated version is available as DEC WRL Research Report 87/2.

[19] Flemming Nielson and Hanne Riis Nielson. Two-level functional languages. *Cambridge Tracts in Theoretical Computer Science*, 34, 1992.

[20] Massimiliano Poletto, Dawson R. Engler, and M. Frans Kaashoek. `tcc`: A template-based compiler for 'C. In *Proceedings of the First Workshop on Compiler Support for System Software*, February 1996.

[21] Calton Pu, Tito Autrey, Andrew Black, Charles Consel, Crispin Cowan, Jon Inouye, Lakshmi Kethana, Jonathan Walpole, and Ke Zhang. Optimistic incremental specialization: Streamlining a commercial operating system. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, December 1995.

[22] Ken Thompson. Regular expression search algorithm. *Communications of the Association for Computing Machinery*, 11(6):419–422, June 1968.