A New Approach to Forth Native Code Generation

M. Anton Ertl

Institut für Computersprachen
Technische Universität Wien
Argentinierstraße 8, A-1040 Wien, Austria
anton@mips.complang.tuwien.ac.at
Tel.: (+43-1) 58801 4459
Fax.: (+43-1) 505 78 38

Abstract. RAFTS is a framework for applying state of the art compiler technology to the compilation of Forth. The heart of RAFTS is a simple method for transforming Forth programs into data flow graphs and static single assignment form. Standard code generation and optimization techniques can be applied to programs in these forms. Specifically, RAFTS uses interprocedural register allocation to eliminate nearly all stack accesses. It also removes nearly all stack pointer updates. Inlining and tail call optimization reduce the call overhead. RAFTS compiles all of Forth, including difficult cases like unknown stack heights, `PICK`, `ROLL` and `EXECUTE`. And last, but not least, RAFTS is designed for interactive Forth systems; it is not restricted to batch compilers.

1 Introduction

Traditionally, Forth is implemented using a threaded code interpreter [Tin81]. This keeps the compiler extremely simple: A word is compiled by appending its address to the current definition. Performance bottlenecks are removed by manually recoding the critical pieces of code in assembly language. While this may be appropriate in embedded systems programming, it does not meet the portability requirements for general-purpose computing.

Another popular avenue to efficiency is the creation of special hardware, which has led to many research-prototype and several commercially available CPUs that are designed to run Forth efficiently [Koo89, HFWZ87]. The code for these stack machines is generated by simple, peephole-optimizing compilers.

Recent research [Ung87, CU89, KB92] proves that unconventional languages can be implemented efficiently on mainstream hardware using aggressive compiler techniques and that the gains of specialized architectures are usually offset by the better device technology available to widely-used RISCs.

The special needs of Forth are fast stack manipulation and fast procedure calls. Optimizing stack manipulation is especially important for RISC processors, where stack accesses cannot be hidden in autoincrement/decrement addressing modes, and usually require two instructions. It turns out that the special requirements of Forth are easier to map into conventional hardware than the requirements of other unconventional languages. However, special attention has to be paid to preserve the interactiveness of Forth.

RAFTS is a framework for applying state of the art compiler technology to Forth compilation, namely interprocedural register allocation (Section 3.5), inlining, tail call optimization (Section 3.3), instruction selection and scheduling (Section 3.1). Several new techniques are introduced that address Forth's special needs: simple methods for transforming Forth programs into data flow graph (Section 3.1) and static single assignment form (Section 3.2), stack pointer update minimization (Section 3.4), treatment of unknown stack heights (Section 3.6), `PICK` and `ROLL` (Section 3.8).

2 Related Work

Parsers can be used transform postfix code into trees, i.e. data flow graphs. But parsers cannot process stack manipulation words and multiple stacks, they cannot produce DAGs and the method of Section 3.1 is simpler.

Stack-based languages have been used as intermediate code for compilers. During code generation, these compilers face problems similar to Forth native code generation[1]. The code generator of the Amsterdam Compiler Kit (ACK) resembles RAFTS in its use of the stack, but it performs all tasks of code generation at the same time [TvSKS83]. Everything else in ACK is different: In contrast to RAFTS, which does not do anything at the Forth level, ACK optimizes as much as possible at the postfix code level, because its main goal is machine-independence.

[1] However, they usually don't have to handle things like `SWAP` and they put a much higher emphasis on (local) variables.

Paper and BibTeX entry are available at http://www.complang.tuwien.ac.at/papers/. This paper was published in: EuroForth '92 Proceedings, pages 73-78

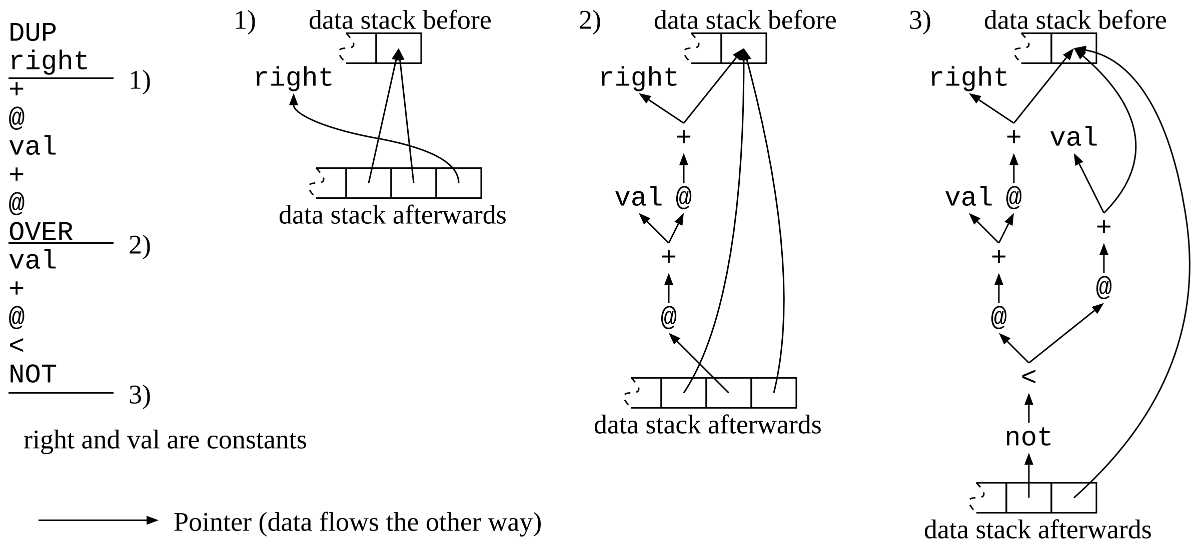

Fig. 1. Building the data flow graph (three snapshots)

There are several Forth compilers that produce machine code [Ros86, Alm86]. Usually they start by generating subroutine-threaded code, then they inline small subroutines, and peephole-optimize the resulting code to remove redundant pushes and pops. Yet much stack manipulation overhead remains. Rose reports that "the resulting machine code is a factor of two or three slower than the equivalent code written in machine language, due mainly to the use of the stack rather than registers."

However, some compilers are pretty sophisticated at reducing stack manipulation overhead. JForth V3.0 can keep up to five values in registers and optimizes words like `SWAP` and `ROT` completely away. This optimization is only performed on sequences of certain primitives. In contrast, RAFTS usually keeps stack items always in registers. Almy's non-interactive (i.e. batch) CFORTH compiler keeps up to two values in registers and reduces stack pointer manipulations [Alm86]. It keeps values in registers across basic block boundaries. However, the lack of global register allocation in his compiler would result in a lot of register shuffling, if more than two registers were available.

3 RAFTS

3.1 Basic Blocks.

Data flow graphs. Basic Blocks consist only of primitives like `+` and `!` (store), of literals, constants and variables, and of stack manipulation words like `SWAP` and `R@`.

Stack manipulation words are compiled by simply executing them. When one of the other words is compiled, the compiler pops and pushes as many items as the word would during execution. But the word is not executed. Instead, the compiler builds a record, which contains the word (to indicate the kind of node) and the operands taken from the stack. A pointer to this data structure is pushed on the stack instead of the result. In this way, the compiler builds a data flow graph[2] (see Fig. 1). This data structure is widely used in compiler construction, so well-known techniques can be used for further compilation.

Further processing. The other three steps of the conversion of a basic block from Forth to native code are instruction selection, instruction scheduling and register allocation. They have been described extensively in the literature, so here they will not be explained in-depth.

Instruction selection combines the operators in the data flow graph into legal instructions of the target machine, transforming the operator DAG into an instruction DAG (see Fig. 2). The state of the art of instruction selection for CISCs is tree parsing automata generated from tree grammars by tools like Burg [FHP91]. For RISCs a much simpler approach suffices: Walking down the data flow graph and combining on the way as many operators as possible is optimal on most RISCs.

All referenced stack items now reside in pseudo-registers (px in Fig. 2), of which an unlimited number can be used [3]; all stack accesses within a basic block have been eliminated. **Register allocation**

[2] For basic blocks the graph is a directed acyclic graph (DAG); compiler textbooks refer to it simply as DAG and draw it up-side-down.

[3] Indeed, in order to make conversion to static single

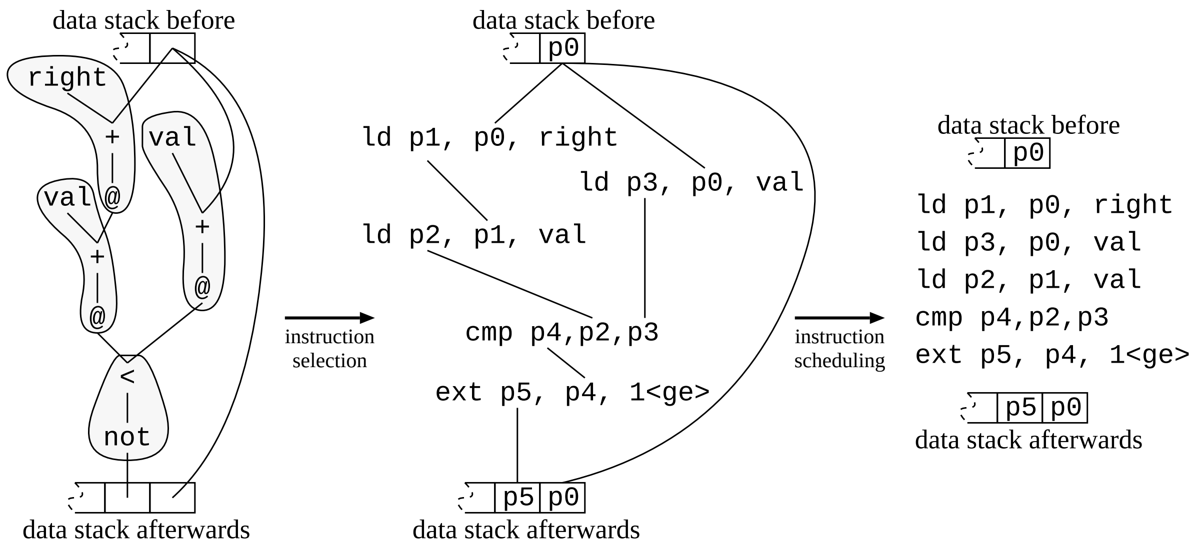

Fig. 2. Instruction selection and scheduling (for the MC88100)

replaces the pseudo-registers with real registers and spills excess pseudo-registers to memory. RAFTS uses interprocedural register allocation, which will be discussed in Section 3.5. In the meantime, the important thing to keep in mind is: Pointers in the instruction DAG correspond directly to pseudo-registers; specifically, at basic block boundaries the pointers on the stacks correspond to pseudo-registers (already before instruction selection).

Instruction scheduling orders the nodes of the instruction DAG, i.e. it transforms the DAG into a list (see Fig. 2). On RISCs the goal of scheduling is to avoid pipeline stalls. The standard algorithm for RISC instruction scheduling is list scheduling [GM86]. On CISCs instructions are scheduled to minimize register pressure. This is usually performed during local register allocation [ASU86].

The nodes of the data flow graph are partially ordered through its edges. But the edges, that RAFTS introduces, are not sufficient, because they reflect only dependences through the stack, but not through memory. Therefore the compiler also keeps track of memory accesses and introduces additional edges between memory accesses and stores.

Details. I left out a few details of data flow graph construction: If the code has input arguments on the stack(s), the compiler will try to access them. Therefore, the compiler must provide a sufficient number of items on the stack. These items stand for the registers where the run-time stack items will reside when the basic block is entered.

Some primitives have several results, which does not fit the data flow graph model well. These words can be divided into two classes: words that really have several results, e.g. `/MOD`, and words with a double precision result. The former can be split into two operations, the latter are handled by pushing pointers to two points in the same record.

There are also a few zero-result primitives, e.g. `!`. If the nodes of the data flow graph were only reachable from the stack, these operations and their predecessors would be lost. While such dead code elimination is desired in cases like `DROP`, stores have to be preserved. Therefore, the compiler remembers them in a special array.

If the compiler is written in Forth, its private use of the stacks must not collide with the use for compilation. This is easy to achieve by using different stacks for different purposes.

3.2 Control structures

Control structures connect basic blocks. This poses the problem of matching the basic blocks.

Control flow splits (`IF`, `WHILE` and `UNTIL`) are easy: The values can stay in the registers they were in. Control flow joins (`ENDIF` and `BEGIN`) are a little harder[4]: The corresponding stack items of the joining basic blocks usually do not reside in the same register.

The compiler now inserts ϕ-functions after the join (see Fig. 3). The ϕ-functions do not correspond to real code (indeed they are removed later); they indicate which values reach the join. The program is now in static single assignment (SSA) form

assignment form trivial, it is essential that no pseudo-register is reused.

[4] Alternatively, we could choose to make joins easy and splits harder by stating that values stay in the same registers during joins.

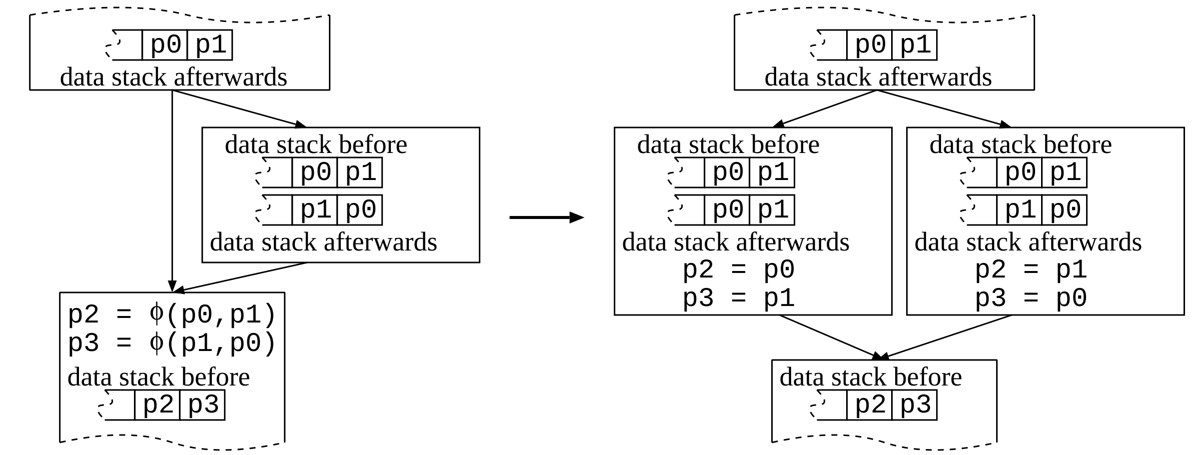

Fig. 3. SSA form of "`IF SWAP ENDIF`" and conversion of ϕ-functions into moves

[CFR+91], which is an excellent representation for analysis and optimization purposes. Optimizations have been treated extensively in the literature and will not be discussed here.

After performing the optimizations, the SSA form can be converted into a more executable form easily: Every ϕ-function is replaced by moves in the parent basic blocks (see Fig. 3). If one of the joining basic blocks has several successors (i.e. it ends with a control flow split), the moves have to be inserted into a new, empty basic block. Many of the inserted moves will be removed by register allocation.

3.3 Calls

Words and their calls are treated like other control structures: A word's entry point is a control flow join, so a consistent state has to be reached. This is again represented by ϕ-functions at the entry point, this time with as many arguments as there are call sites. Later they will be converted into moves at the call sites. Like other control flow splits, the `EXIT`s pose no problem, unless there are several of them. If a word has several `EXIT`s, they have to be treated like a control flow join just before exiting.

Inlining has been the starting point of most Forth native code compilers. In the framework of RAFTS, it can eliminate calling and returning overhead and stack pointer updates, it can improve register allocation and reveal opportunities for further optimization. The important decision in inlining is what to inline [CMCH92].

If the last useful (i.e. non-stack-manipulation) word is a call, **tail call optimization** can be applied: The call is converted into a jump and the called word returns directly to its callers caller. Of course, this optimization must not be performed if the called word does nasty things to its return address (e.g. throwing it away).

3.4 Stack pointer updates

Nearly all stack pointer updates can be optimized away: The compiler just remembers by how much the physical stack pointers differ from the logical stack pointers and updates this offset at compile-time. There are only a few situations when physical stack pointer updates have to be generated: A word may be called from several sites which usually have different offsets. The compiler sets the word's entry offsets to the offsets of one call site. At the other call sites, the stack pointers have to be updated before the call. This may result in different offsets at control flow joins. So before such a join another round of pointer updates has to be generated to make the offsets equal.

3.5 Register allocation

Given the high frequency of calls (12% of the executed primitives [Koo89]), interprocedural register allocation is necessary to make effective use of the register set. Otherwise most of the time would be spent saving and restoring registers at procedure entries and exits, and around calls.

There are a number of good register allocation algorithms, but it is not yet clear which one is the most appropriate: It should perform good global and interprocedural allocation, but, in order to preserve the interactive feeling, it must not take a long time.

Graph coloring register allocation [Cha82, CH90, Bri92], currently the standard approach, needs a lot of time and space. Hierarchical graph coloring [CK91] looks better. Other alternatives are coagulation [Mor91] and interprocedural allocators [Cho88].

Where to spill. In the unlikely[5] event that more registers are needed than provided by the processor, some pseudo-registers have to be spilled to memory. In conventional compilers they are spilled to their home locations on the stack. But, in Forth, this home location can be changed by stack operations. Nevertheless, a pseudo-register can be spilled to any free location on any of the stacks. All locations belonging to register allocated stack items and locations beyond the logical stack pointers are free.

3.6 Unknown stack height

The stack height of a piece of code is unknown, if it cannot be determined at compile time. Unknown stack heights are rare (except `?DUP`), but they occur. The compiler can detect them by comparing the logical stack pointers at control flow joins. If they are different, the stack height becomes unknown[6].

RAFTS compiles unknown stack heights by forcing the stack heights of the joining basic blocks to become the equal: One of the joining basic blocks is chosen as the standard and the other one is adapted to fit the standard by inserting code for stack pointer adjustment and the respective load(s) or store(s). If a loop is involved, the adjustment code has to be inserted into the loop. Since the adjustment changes the position of the spilled values (with respect to the logical stack pointer), they may have to be moved in order to make the state consistent at the control flow join.

The only common variable stack effect word is `?DUP` in contexts like `?DUP 0= IF`, where the `IF` has a variable stack effect, too. However, the combined stack effect is usually fixed. The compiler can recognize and handle this situation with ad-hoc logic.

3.7 `EXECUTE`

When compiling `EXECUTE`, the compiler does not know what word will be called and therefore cannot perform interprocedural register allocation. Even worse, the stack effect of the executed word is unknown.

Therefore, when compiling `EXECUTE`, the state is adapted to a calling convention: A number of items from each stack has to reside in certain registers. The rest is stored in the appropriate places on the stacks; the physical stack pointers have to be updated to reflect their logical values. It will have to be determined empirically, how many items of each stack should reside in registers.

[5] see [HFWZ87, Koo89] for an empirical stack usage analysis

[6] The compiler could warn the user, since unknown stack heights are often caused by programming errors [Tev89].

When the address of a word is taken, a special version of the word is generated that conforms to the calling convention.

3.8 `PICK` and `ROLL`

Compiling `PICK` and `ROLL` with constant top-of-stack (e.g. `4 PICK`) poses no problem. They can be executed during compilation like other stack manipulation words. But if the top-of-stack value cannot be determined at compile time, this is not possible. There are two solutions:

- The registers are dumped on the stack and the operation is performed. In case of `ROLL`, the compiler also has to invalidate the registers and reload the stack items on demand.
- They are translated into something like

```
CASE
  0 OF  0 PICK  ENDOF
  1 OF  1 PICK  ENDOF
  ...
  DUP PICK SWAP
ENDCASE
```

  with one `OF` for every stack item that resides in a register.

3.9 How it fits together

During the processing of the input text, the compiler builds the data flow graphs for the basic blocks and control flow graphs for words. Only the words lower in the call graph have been built yet; therefore, not enough interprocedural information is available and the compiler leaves the rest of the work for a later stage.

The rest of compilation is started by the first call to an uncompiled word. Then this word and all words that it calls are compiled: Unknown stack heights, `EXECUTE`, `PICK` and `ROLL` are resolved, stack pointer updates are inserted, and the code is converted to SSA form. After the optimizations, the code is converted back from SSA form. Then instruction selection is performed. Instruction scheduling and register allocation may be performed in any order. Finally, the code represented by the data structures of the compiler is translated into real code and the word is executed.

4 Further Work

The proof of the pudding is in its eating. RAFTS has to be implemented and evaluated through measurements. To make the resulting system a real, complete Forth system decompiling and debugging facilities must be developed.

Can the techniques presented here be applied to Postscript? How profitable are they? Since Postscript performs run-time type-checking, a Postscript compiler would also need techniques for reducing that overhead, e.g. those developed by the Self group [CU89].

The surprising ease, with which convenient representation like data flow graph and SSA form can be produced from Forth code, inspires the question, what other nice properties stack languages in general and Forth in particular may have.

Acknowledgements

Felix Beer, Manfred Brockhaus, Andreas Krall, Herbert Pohlai and Franz Puntigam provided useful comments on draft versions of this paper. Tom Almy, Mike Haas, Mike Hore and Bernd Paysan discussed their native code compilers with me.

References

[Alm86] Thomas Almy. Compiling Forth for performance. *Journal of Forth Application and Research*, 4(3):379–388, 1986.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers. Principles, Techniques, and Tools.* Addison-Wesley, 1986.

[Bri92] Preston Briggs. *Register Allocation via Graph Coloring.* PhD thesis, Rice University, Houston, 1992.

[CFR+91] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[CH90] Fred C. Chow and John L. Hennessy. The priority-based coloring approach to register allocation. *ACM Transactions on Programming Languages and Systems*, 12(4):501–536, October 1990.

[Cha82] G. J. Chaitin. Register allocation & spilling via graph coloring. In *SIGPLAN '82 Symposium on Compiler Construction*, pages 98–105, 1982.

[Cho88] Fred C. Chow. Minimizing register usage penalty at procedure calls. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 85–94, 1988.

[CK91] David Callahan and Brian Koblenz. Register allocation via hierarchical graph coloring. In *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 192–203, Toronto, 1991.

[CMCH92] Pohua P. Chang, Scott A. Mahlke, William Y. Chen, and Wen-mei W. Hwu. Profile-guided automatic inline expansion for C programs. *Software—Practice and Experience*, 22(5):349–369, May 1992.

[CU89] Craig Chambers and David Ungar. Customization: Optimizing compiler technology for SELF, a dynamically-typed object-oriented programming language. In *SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages 146–160, 1989.

[FHP91] Christopher W. Fraser, Robert R. Henry, and Todd A. Proebsting. BURG — *Fast Optimal Instruction Selection and Tree Parsing*, 1991. Available via anonymous `ftp` from `kaese.cs.wisc.edu`, file `pub/burg.shar.Z`.

[GM86] Phillip B. Gibbons and Steve S. Muchnick. Efficient instruction scheduling for a pipelined architecture. In *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction*, pages 11–16, 1986.

[HFWZ87] John R. Hayes, Martin E. Fraeman, Robert L. Williams, and Thomas Zaremba. An architecture for the direct execution of the Forth programming language. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS-II)*, pages 42–48, 1987.

[KB92] Andreas Krall and Thomas Berger. Fast Prolog with a VAM_{1p} based Prolog compiler. In *Programming Language Implementation and Logic Programming (PLILP '92)*, pages 245–259. Springer LNCS 631, 1992.

[Koo89] Philip J. Koopman, Jr. *Stack Computers.* Ellis Horwood Limited, 1989.

[Mor91] W. G. Morris. CCG: A prototype coagulating code generator. In *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 45–58, Toronto, 1991.

[Ros86] Anthony Rose. Design of a fast 68000-based subroutine-threaded Forth with inline code & an optimizer. *Journal of Forth Application and Research*, 4(2):285–288, 1986. Proceedings of the 1986 Rochester Forth Conference.

[Tev89] Adin Tevet. Symbolic stack addressing. *Journal of Forth Application and Research*, 5(3):365–379, 1989.

[Tin81] C. H. Ting. *Systems Guide to fig-Forth.* Offete Enterprises, Inc., San Mateo, CA 94402, 1981.

[TvSKS83] Andrew S. Tanenbaum, Hans van Staveren, E. G. Keizer, and Johan W. Stevenson. A practical tool kit for making portable compilers. *Communications of the ACM*, 26(9):654–660, September 1983.

[Ung87] David Ungar. *The Design and Evaluation of a High-Performance Smalltalk System.* MIT Press, 1987.