PAF: A portable assembly language based on Forth

M. Anton Ertl*
TU Wien

Abstract

A portable assembly language provides access to machine-level features like memory addresses, machine words, code addresses, and modulo arithmetics, like assembly language, but abstracts away differences between architectures like the assembly language syntax, instruction encoding, register set size, and addressing modes. Forth already satisfies a number of the characteristics of a portable assembly language, and is therefore a good basis. This paper presents PAF, a portable assembly language based on Forth, and specifically discusses language features that other portable assembly languages do not have, and their benefits; it also discusses the differences from Forth. The main innovations of PAF are: tags indicate the control flow for indirect branches and calls; and PAF has two kinds of calls and definitions: the ABI ones follow the platform's calling convention and are useful for interfacing to the outside world, while the PAF ones allow tail-call elimination and are useful for implementing general control structures.

1 Introduction

Traditionally compilers have produced the assembly language for the various target architectures, and interpreters were written in assembly language. The disadvantage of this approach is that it requires retargetting for every new architecture. As a result, many such compilers and interpreters target only one or few architectures, and ports to new architectures often take quite a while.[1]

Portable assembly languages promise to solve this problem: Compile to (or implement the interpreter in) the portable assembly language, and the compiler or interpreter will work on a variety of architectures without extra effort. Of course the portable assembly language implementation has to be targeted for these architectures, but that effort can be reused (and possibly the cost shared) by several compilers/interpreters.

In this paper we present a new portable assembly language, PAF (for "Portable Assembly Forth"). There have been a number of languages that been designed and/or used as portable assembly languages (Section 2), so why introduce a new one?

1.1 Contributions

An issue that a number of them have had is that they require the code to be organized in functions that follow the standard calling convention (ABI) of the platform, which usually prevents tail-call optimization. PAF provides ABI calls and definitions for interfacing with the rest of the world, but also PAF calls and definitions, which (unlike ABI calls) can be tail-call-optimized and can therefore be used as universal control flow primitives [Ste77] (see Section 3.9 and 3.10).

*Correspondence Address: Institut für Computersprachen, Technische Universität Wien, Argentinierstraße 8, A-1040 Wien, Austria; `anton@mips.complang.tuwien.ac.at`

[1] E.g., AMD64 CPUs became available in 2003; The lina interpreter for AMD64 became available in 2008, the iForth compiler became available for AMD64 in 2009 (and 32-bit releases were stopped at the same time), and other significant Forth compilers like SwiftForth, VFX, and bigForth still do not offer 64-bit support in 2013. By contrast, the Gforth interpreter which uses a portable assembly language, was available there right from the start (thanks to our portable assembly language being there from the start), and we verified that by building and testing Gforth on an AMD64 system in August 2003.

Paper and BibTeX entry are available at http://www.complang.tuwien.ac.at/papers/. This paper was published in: Kolloquium Programmiersprachen (KPS'13)

Another problem is that indirect branches and calls have a high cost, because the compiler has to assume that every branch/call can reach any entry point. PAF introduces tags to specify which branches/calls can reach which entry points (see Section 3.9 and Section 3.9).

The most significant difference between PAF and Forth is that PAF contains restrictions that ensure that the stack depth is always statically determinable, so stack items can be mapped to registers. It is interesting that these restrictions are relatively minor and don't affect much Forth code; it's also interesting to see an example of Forth code that is affected (see Section 5).

2 Previous Work

This section discusses existing portable assembly languages, their features and deficiencies and why we feel the need for a new one.

2.1 C

C and its dialects, like GNU C, have been used as a portable assembly language in many systems: It is the prevalent language for writing interpreters (e.g., Python, Ruby, Gforth) and run-time systems; C has also been used as target language for compilers: (e.g., the original C++ compiler cfront, and one of the code generation options of GHC).

However, the C standard specifies a large number of "undefined behaviours", including things that one expects to behave predictably in a portable assembly language, e.g., signed integer overflow. In earlier times this was not a problem, because the C compilers still did what the programmer intended. Unfortunately, a trend in recent years among C compiler writers has been to "optimize" programs in such a way that it miscompiles (as in "not what the programmer intended") code that earlier compiler versions used to compile as intended. While it is usually possible to find workarounds for such a problem, the next compiler version often produces new problems, and with all these workarounds the direct relation from language feature to machine feature is lost.

Another problem of C (and probably a reason why it is not used as often as compiler target language as for interpreters) is that its control flow is quite inflexible: Code is divided into C functions, that can be called and from which control flow can return; the only other way to change control flow across functions is `longjmp()`.

Varargs in combination with other language features has led to calling conventions where the caller is responsible for removing the arguments from the stack. This makes it impossible in to implement guaranteed tail-call optimization, which would be necessary to use C calls as a general control flow primitive [Ste77].

As a result, any control flow that does not fit the C model, such as unlimited tail calls, backtracking, coroutining, and even exceptions is hard to map to C efficiently.

2.2 LLVM

LLVM is an intermediate representation for compilers with several front ends, optimization passes and back ends [LA04].

Unfortunately, it shares many of the problems of C: In particular, you have to divide the code into functions that follow some calling convention, restricting the kind of control flow that is possible. To work around this problem, it is possible to add your own calling convention, but that is not easy.[2]

LLVM was also promised to be a useful intermediate representation for JIT compilers, but reportedly its code generation is too slow for most JIT compiler uses.

LLVM supports fewer targets than C. Given that it also seems to share many of the disadvantages of C, it does not appear to be an attractive portable assembly language to me, despite the buzz it has generated.

[2] Usenet message `<KYGdnTH8PMyMpM7MnZ2dnUVZ_j-dnZ2d@supernews.com>`

2.3 C--

C-- [JRR99] has been designed as portable assembly language. Many considerations went into its design, and it appears to be well-designed, if a little too complex for my taste, but the project appears to be stagnant as a general portable assembly language, and it seems to have become an internal component of GHC (called Cmm there).

While C-- does not appear to be an option as portable assembly language for use in practical projects at the moment, looking at its design for inspiration is a good idea.

2.4 Vcode and GNU Lightning

Vcode [Eng96] is a library that provides a low-level interface for generating native code quickly (10 executed instructions for generating one instruction) and portably. It was part of a research project and has not been released widely, but it inspired GNU Lightning, a production system.

The demands of extremely fast code generation mean that GNU Lightning cannot perform any register allocation on its own. Therefore the front end has to perform the register allocation. It also does not perform instruction selection; each Lightning instruction is translated to at least one native instruction.

GNU Lightning also divides the code into functions that follow the standard calling convention, and one can call functions according to the calling convention. However, it is also possible to implement your own calling conventions and other control flow, because the front end is in control of register allocation, but (from reading the manual) it is not clear if this can be integrated with the stack handling by GNU Lightning und if one can use the processor's call instruction for your own calling convention.

It is possible to use better code generation technology with the GNU Lightning interface, and also to provide ways to use the processor's call and return instructions for your own calling convention.

With these changes, wouldn't the GNU Lightning interface be the perfect portable assembly language? It would certainly satisfy the basic requirements of a portable assembly language, but as a replacement for a language like C, it misses conveniences like register allocation.

3 Portable Assembly Forth (PAF)

3.1 Goals

- Portability: Works on several different architectures
- Direct relation between language feature and machine feature, i.e., if you look at a piece of PAF code, you can predict what the machine code will look like.

  However, the relation between PAF and the machine is not as direct as for GNU Lightning: There is register allocation and instruction selection, there may be instruction scheduling, and code replication. Instruction selection and instruction scheduling make better code possible (at the cost of slower compilation); register allocation interacts with these phases, and leaving it to the clients would require duplicated work in the clients, as register allocation is not really language-specific.
- Capabilities of the (user-mode part of the) machine can be expressed in PAF. However, this goal is moderated by the needs of clients and by the portability goal. I.e., PAF will at first only have language features that compilers and interpreters are likely to need (features can be added when clients need them); and machine features of particular architectures that cannot be abstracted into a language feature that can be implemented reasonably on the intended target machines will not be supported, either.

3.2 Target machines

While a portable assembly language can abstract away some of the differences between architectures, there are differences that are too difficult to bridge, and would lead PAF too far away from the idea of a direct correspondence between language feature and architectural feature, so here we define the class of machines that we target with PAF:

PAF targets general-purpose computer architectures, i.e., the architectures that have been designed as compiler targets, such as AMD64, ARM, IA-32, IA-64, MIPS, PowerPC, SPARC.

Memory on the target machines is byte-addressed with a flat address space; e.g., DSPs with separate X and Y address spaces are not target machines. The target machines use modulo (wrap-around) arithmetics and and signed numbers are represented in 2s-complement representation.

The target machines have a uniform register set for integers and addresses (not, e.g., accumulators with different size than address registers), and possibly separate (but internally also uniform) floating point registers.

3.3 Forth and PAF

Forth's low-level features are quite close to assembly language; e.g., like in assembly language, neither the compiler nor the run-time system maintains a type system, and the language differentiates between different operations based on name, not based on type; e.g., Forth has `<` for signed comparison and `U<` for unsigned comparison of cells (machine words), just like MIPS has `slt` and `sltu`, and Alpha has `cmplt` and `cmpult`.

Therefore Forth is a good basis for a portable assembly language. However, there are features that are problematic in this context: In particular, in Forth the stack depth is not necessarily statically determined (unlike in the JVM), even though in nearly all Forth code the stack depth is actually statically determined (known to the programmer, but not always the Forth system). So we change these language features for PAF.

A number of higher-level features of Forth are beyond the goal of a portable assembly language, so PAF does not support them.

On the other hand, there are a few things that are missing in standard Forth that have to be added to PAF, such as words for accessing 16-bit quantities in memory.

3.4 Example

The following example shows two definitions written in PAF:

```
                     \  cmpl %edx,%eax
: max                \  jle L28
  2dup >? if         \  ret
    drop exit endif  \ L28:
  nip exit ;         \  movl %edx,%eax
                     \  ret
abi:xx- printmax {: n1 n2 -- :}
  "max(%ld,%ld)=%ld\n\0" drop
  n1 n2 2dup max abi.printf.xxxx-
  exit ;

\ Call from C:
\ main() { printmax(3,5); return 0; }
```

The first, `max`, looks almost like conventional Forth code, and corresponding assembly language code for IA-32 is shown in comments to the right. `max` does not have a fixed calling convention; the PAF compiler can set a calling convention that is appropriate for `max` and its callers (e.g., it can be tail-called). Since `max` does not follow the platform's calling convention, it cannot be called from, e.g., C code.

The second definition, `printmax`, follows the standard ABI of the platform (as indicated by using an `abi:` defining word. The `xx-` in `abi:xx-` shows that `printmax` expects and consumes two cells from the data stack and 0 floats from the FP stack and produces 0 cells and 0 floats; a C prototype for this definition could be `void printmax(long, long)`. `Printmax` calls `max`, and the compiler can choose the calling interface between the call and max; it calls `printf` using the standard calling convention with the call `abi.printf.xxxx-`, where the `xxxx-` indicates that four cells are passed as integer/address parameters and the return value of printf is ignored.

Locals are used in `printmax` but can be used in every definition. `Exit`ing from the definitions is explicit.

3.5 Registers

Several language features correspond to real machine registers: Stack items, locals, and values.

Stack items (elements) are useful for relatively short-lived data and (unlike locals) can be used for passing arguments and return values. There is no stack pointer and memory area specific to the stack, it's just an abstraction used by the compiler. Stack manipulation words like `DUP` or `SWAP` just modify the data flow and there is no machine code that directly corresponds to them (indirect consequences may be, e.g., move instructions at control flow joins).

Locals live within a definition and are a convenience: Local variables of the source language can be mapped directly to PAF's locals without needing register allocation or stack management in the front end. If a source local needs to be distributed across several PAF definitions (e.g., because a control structure of the source language is mapped to a PAF (tail) call), the local can be defined in each of these definitions, and the contants are passed on the stack across calls; this is not as convenient as one might like, but seems to be a good compromise.

Values are global (thread-local) variables whose address cannot be taken, so they can be stored in registers.

If stack items and locals don't fit in the registers, they are stored in a stack that is not visible to PAF code; this stack stores items from the stack[3], locals, and return addresses, so this does not correspond to the memory representation of, e.g., the data stack.[4]

If values don't fit in the registers, they are stored in global/thread-local memory.

3.6 Memory

The words `c@ uw@ ul@ ( addr -- u )` load unsigned 8/16/32-bit values from memory, while `sc@ w@ l@ ( addr -- n )` load signed 8/16/32-bit values from memory; `@ (addr -- w)` loads a cell (32-bit or 64-bit, depending on the machine) from memory; `sf@ df@ ( addr -- r )` load 32/64-bit floating-point values from memory. `c! w! l! ! ( x addr -- )` and `sf! df! ( r addr -- )` store stack items to memory.

3.7 Arithmetics

The usual Forth words `+ - * negate and or invert lshift rshift` correspond to the arithmetic and logic instructions present in every machine. There are also additional words like `/ m* um* um/mod sm/rem` that correspond to instructions on some machines, and have to be synthesized from other instructions on other machines.

[3]More precisely, the data stack for cells and the floating-point stack for floating-point numbers, but that detail plays no further role in this paper.

[4]Some languages have local variables whose address can be taken; it may be a good idea to provide a way to store them in this stack eventually, but for now such variables have to be stored elsewhere. The interaction of such a feature with, e.g., tail calls has to be considered first.

3.8 Comparison

The words `=? <? u<? f=? f<?` etc. compare two stack items and return 0 for false and 1 for true. They correspond to the Forth words `= < u< f= f<` etc., with the difference that the Forth words return −1 (all-bits-set) for true. A number of machines have instructions that produce 0 or 1 (MIPS, Alpha, IA-32, AMD64), while for others it is as easy to produce 0 or 1 as to produce 0 or −1, so "0 or 1" is more in line with the goal of the direct relation to the machine feature. An implementation of a 0-or-−1 language like Forth would use a sequence like `<? negate` for which good code can be generated easily.[5]

3.9 Control flow

... inside definitions

The standard Forth words `begin again until ahead if then cs-roll` are available in PAF and are useful for building structured control flow, such as `if ... then ... elsif ... then ... else ... end`.

While one can construct any control flow with these words [Bad90], if you want to implement labels and gotos, it's easier to use labels and gotos. Therefore, PAF (unlike Forth) provides that, too: `L:`*`name`* defines a label and `goto:`*`name`* jumps to it.

PAF also supports indirect gotos: *`'name/tag`* produces the address of label name, and `goto/`*`tag`* jumps to it. The tag indicates which gotos can jump to which labels; a PAF program must not jump to a label address generated with a different tag. E.g., a C compiler targeting PAF could use a separate tag for each `switch` statement and the labels occuring there.

These tags are useful for register allocation. One can use different tags when taking the address of the same label several times, and this may result in different label addresses, with the code at each target address matched to the gotos that use that tag (i.e., several entry points for the same PAF label).

Whichever method of control flow you use, on a control flow join the statically determined stack depth has to be the same on all joining control flows. This ensures that the PAF compiler can always determine the stack depth and can map stack items to registers even across control flow. This is a restriction compared to Forth, but most Forth code conforms with this restriction. Breaking this rule can be detected and reported by the PAF compiler.

PAF Definitions and PAF calls

A definition where the compiler is free to determine the calling interface is defined in the classical Forth way:

```
: name ... exit ;
```

The end of the definition does not produce an implicit return (unlike Forth), so you have to return explicitly with `exit`.

You call such a definition by writing its name, i.e., the traditional Forth way. You can explicitly tail-call such a definition with `jump:`*`name`*; this can be written explicitly, in the spirit of having a portable assembly language. Optimizing implicit tail calls is not hard, so the PAF compiler may do it, too.

We can take the address of a definition with *`'name:tag`*, call it with `exec.`*`tag`* and tail-call it with `jump.`*`tag`*. The tags indicate which calls can call which definitions.

The stack effects of all definitions whose address is taken with the same tag have to be compatible. I.e., there must be one stack effect that describes all of them; e.g., `( x x -- x )` is a valid stack effect of both `+` and `drop` (although the minimal stack effect of `drop` is `( x -- )`), so `+` and `drop` have compatible stack effects.

[5] Conversely, one might also decide to have `<` etc. instead of `<?` in PAF, and let the compiler handle the mismatch to some machines, but that would be somewhat against the spirit of a portable assembly language.

The use of tags here has two purposes: It informs the PAF compiler about the control flow; and it also informs it about the stack effect of the indirect call (while a Forth compiler usually has to assume that EXECUTE can call anything, and have any stack effect).

3.10 ABI definitions and ABI calls

We need to specify the stack effect explicitly as signature of an ABI definition or call. The syntax for such a signature is `[xr]*-[xr]*`, where x indicates a cell (machine word/integer/address) argument, and r a floating-point argument; the letters before the `-` indicate parameters, and the letters afterwards the results. The division into `x` and `r` reflects the division into general-purpose registers and floating-point registers on real machines, and the role these registers play in calling conventions.

A definition conforming to the calling convention is defined with `abi:`*`sig name`*. *Sig* specifies the stack effect, and indicates the correspondence between ABI parameters and PAF stack items. This signature is not quite redundant, e.g., consider the difference between the following definitions:

```
abi:x-x id   exit ;
abi:-   noop exit ;
```

These definitions differ only in the signature, yet they behave differently: `id` returns its argument, `noop` doesn't, and with ABI calling conventions, there is usually a difference between these behaviours.

You can call to an ABI-conforming function with `abi.`*`name.sig`*, where name is the name of the function (which may be a PAF definition or a function written in a different language and dynamically or statically linked with the PAF program). The signature specifies how many and which types of stack items to pass to the called functions, and what type of return value (if any) to push on the stack.

Putting the signature on every call may be a bit repetetive for human programmers, but PAF is mainly intended as an intermediate language, and an advantage of this scheme is that different calls to the same function (e.g., `printf`) can have different stack effects.

You can take the address of an ABI funtion with `abi'`*`name`* and call it with `abi-exec.`*`sig`*. There are no tail calls to ABI functions, because we cannot guarantee that tail calls can be optimized in all calling conventions.

Unlike PAF definitions, for ABI functions there is no point in tagging these function addresses, because the call always uses the ABI calling convention (whereas the compiler is free to determine the calling interface for PAF calls). The signature in indirect ABI calls has the same significance as in direct ABI calls.

3.11 Discussion

Why have two kinds of definitions and two kinds of calls?

The PAF definitions and calls allow to implement various control structures such as backtracking through tail calls [Ste77]. They also allow the compiler to use flexible and possibly more efficient calling interfaces than the ABI calling convention.

On the other hand, the ABI counterparts allow interfacing with other languages and using dynamically or statically linked binary libraries, including callbacks, and using PAF to build such libraries (e.g., as plug-ins).

4 Non-Features

This section discusses various features that PAF does not have and why.

4.1 Garbage collection

A number of virtual machines, e.g., the Java VM, support garbage collection. However, this feature significantly restricts what can be done. In particular, the data representations are restricted, and one cannot implement "unmanaged" languages or use a different data representation for a garbage collected language (e.g., the Java VM representation is quite different from how most Prolog or Lisp systems represent their data).

Even C--, which is intended as a portable assembly language for garbage collected languages does not implement garbage collection itself, but leaves it to the higher-level language, because that leaves the full freedom on how to implement data and garbage collection to the higher-level language [JRR99].

4.2 Types

PAF does not perform type checking during compilation, nor at run-time; also, there is no overloading of several operations on the same operator based on types. This is consistent with the descent from Forth, and non-portable assembly languages have the same approach.

However, in C-- the compiler knows about data types and uses that knowledge for overloading resolution. The disadvantage of such approaches is that it complicates the C-- compiler without making life easier for the front end compiler, which has to know exactly anyway whether it wants to perform, say, signed or unsigned comparison.

One may wonder about the "absence" of some operations in PAF; e.g., there is `<` `U<`, but only `=` `+` `-` `*`. The reason is that, on the two's-complement machines that PAF targets, these operations are the same for signed and unsigned numbers.

4.3 Debugger

Quite a bit of effort in C-- is devoted to supporting the standard debugger. For now there are no plans to make such an effort for PAF. C became a successful portable assembly language even though it has very little debugger support for languages that use it as intermediate language.

4.4 SIMD

Supporting SIMD instruction set extensions such as SSE, AVX, AltiVec etc. is not planned, mainly because few higher-level languages need such features. They can be added later if there is demand.

5 PAF vs. Forth

Some of the differences between PAF and Forth provide new insights into certain language features, and we take a closer look at that in this section.

The most interesting difference between PAF and Forth is in dealing with the stacks: PAF has restrictions and features that allow the compiler to statically determine the stack depth.

As a consequence, in PAF there is no need to implement the stacks in memory, with a stack pointer for each stack (data stack and return stack for cells, floating-point stack for floating-point values). In contrast, Forth needs to have a separate memory area and stack pointer for each stack, and while stack items can be kept in registers for most of the code, there are some words (in particular, `EXECUTE`) and code patterns (unbalanced stack effects on control flow joins), that force stack items into memory and usually also force stack pointer updates.

This property of Forth is avoided in PAF by requiring balanced stack effects on control flow joins (see Section 3.9), and by replacing `EXECUTE` with `exec.`*`tag`* (see Section 3.9); all definition addresses returned for a particular tag are required to have compatible stack effects, so `exec.`*`tag`* has a statically determined stack effect.

The effect on programs is relatively small: most Forth code has balanced stack effects for control flow anyway, and most occurences of `'` and `execute` can be converted to their tagged

```
\ Forth
: selector ( offset -- )
  create ,
does> ( ... o -- ... )
  @ over @ + @ execute ;

1 cells selector foo
2 cells selector bar

\ PAF
: foo ( ... o -- ... )
  dup @ 1 cells + @ jump.foo ;
: bar ( ... o -- ... )
  dup @ 2 cells + @ jump.bar ;
```

Figure 1: Defining method selectors in Forth and in PAF (simplified)

variants, because programmers keep the stack depth statically determinable in order to keep the code understandable.

However, there are cases where the restrictions are not so easy to comply with. E.g., object-oriented packages in Forth use EXECUTE for words with arbitrary stack effects. Programs using these words have a statically determined stack effect, too, but it is only there at a higher level; e.g., if you use a separate tag (and a separate `exec.`*`tag`*) for each method selector, typical uses would comply with the restriction, but in most object-oriented packages there is only one `execute`.

Figure 1 shows code for this example: the Forth variant defines a defining word `selector`, and the selectors are then defined with this defining word; in contrast, the PAF variant defines the selectors directly (and pretty repetetively), each with its own tag.

If you want to define a defining word for method selectors like you usually do in Forth, the tag would have to be passed around as a define-time parameter between the involved defining words. This support for higher-level programming is not required inside PAF (there we leave such meta-programming to the higher-level language), but if we want to transfer the tag idea back to Forth, we would have to do such things.

6 Related work

We have discussed C, LLVM, C--, and Vcode/GNU Lightning in Section 2.

There are projects that are similar to PAF in using a restricted or modified form of a higher-level language as portable assembler:

- The Python system PyPy uses a restricted form of Python called RPython as low-level intermediate language [AACM07].
- Asm.js[6] is a subset of JavaScript that is so restricted that it can serve as portable assembly language.
- PreScheme is a low-level subset of Scheme used as intermediate language for implementing Scheme48 [KR94].

In all these cases the base language is much higher-level than Forth, and it is much more of a stretch to create a low-level subset than for Forth..

Machine Forth (which evolved into colorForth) is a simple variant of Forth created by Chuck Moore, the inventor of Forth. It closely corresponds to the instructions on his Forth CPUs, but

[6] `http://asmjs.org/`

he also wrote an implementation for IA-32 that creates native code. The IA-32 compiler is very simple, basically just expanding the words into short machine code sequences.[7] It does not map stack items beyond the top-of-stack to registers, yet the generated code is relatively compact; this reflects the fact that machine Forth is close to the machine, including IA-32.

7 Conclusion

PAF is a subset/dialect of Forth that is intended as a portable assembly language. The main contributions of PAF are:

- *Tags* to indicate which indirect branches can reach which labels and which indirect calls can call which definitions. This restricts register allocation far less than unrestricted indirect branches and calls; and it needs less effort, and produces better results than trying to achieve the same result through program analysis.
- The division of definitions and calls into those conforming to the ABI/calling convention of the machine, and others for which the compiler can use any calling interface (and different ones for different sets of callers and callees). In particular, this allows tail-call optimization (unlike ABI calling conventions), which in turn means that we can use the calls as a primitive for arbitrary control structures (e.g., coroutining).
- Restrictions (compared to Forth) on the use of stack items that make it possible to have a static relation between stack items and registers for all programs, and avoid the need for a separate stack pointer and memory area for each stack. This highlights which Forth features are expensive and where they are used.

References

[AACM07] Davide Ancona, Massimo Ancona, Antonio Cuni, and Nicholas D. Matsakis. RPython: a step towards reconciling dynamically and statically typed OO languages. In Pascal Costanza and Robert Hirschfeld, editors, *DLS*, pages 53–64. ACM, 2007.

[Bad90] Wil Baden. Virtual rheology. In *FORML'90 Proceedings*, 1990.

[Eng96] Dawson R. Engler. VCODE: A retargetable, extensible, very fast dynamic code generation system. In *SIGPLAN '96 Conference on Programming Language Design and Implementation*, pages 160–170, 1996.

[JRR99] Simon L. Peyton Jones, Norman Ramsey, and Fermin Reig. C--: a portable assembly language that supports garbage collection. In *International Conference on Principles and Practice of Declarative Programming*, September 1999.

[KR94] Richard A. Kelsey and Jonathan A. Rees. A tractable Scheme implementation. *Lisp and Symbolic Computation*, 7(4):315–335, 1994.

[LA04] Chris Lattner and Vikram S. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Code Generation and Optimization (CGO)*, pages 75–88. IEEE Computer Society, 2004.

[Ste77] Guy Lewis Steele Jr. Debunking the "expensive procedure call" myth or procedure call implementations considered harmful or lambda: The ultimate goto. AI Memo 443, MIT AI Lab, October 1977.

[7] `http://www.colorforth.com/forth.html`