Future Generation Computer Systems 16 (2000) 739–751

Abstract machines for programming language implementation

Stephan Diehl[a,*], Pieter Hartel[b], Peter Sestoft[c]

[a] *FB-14 Informatik, Universität des Saarlandes, Postfach 15 11 50, 66041 Saarbrücken, Germany*
[b] *Department of Electronics and Computer Science, University of Southampton, Highfield, Southampton SO17 1BJ, UK*
[c] *Department of Mathematics and Physics, Royal Veterinary and Agricultural University, Thorvaldsensvej 40, DK-1871 Frederiksberg C, Denmark*

Accepted 24 June 1999

Abstract

We present an extensive, annotated bibliography of the abstract machines designed for each of the main programming paradigms (imperative, object oriented, functional, logic and concurrent). We conclude that whilst a large number of efficient abstract machines have been designed for particular language implementations, relatively little work has been done to design abstract machines in a systematic fashion. © 2000 Elsevier Science B.V. All rights reserved.

Keywords: Abstract machine; Compiler design; Programming language; Intermediate language

1. What is an abstract machine?

Abstract machines are *machines* because they permit step-by-step execution of programs; they are *abstract* because they omit the many details of real (hardware) machines.

Abstract machines provide an intermediate language stage for compilation. They bridge the gap between the high level of a programming language and the low level of a real machine. The instructions of an abstract machine are tailored to the particular operations required to implement operations of a specific source language or class of source languages.

Common to most abstract machines are a program store and a state, usually including a stack and registers. The program is a sequence of instructions, with a special register (the program counter) pointing at the next instruction to be executed. The program counter is advanced when the instruction is finished. This basic control mechanism of an abstract machine is also known as its execution loop.

1.1. *Alternative characterizations*

The above characterization fits many abstract machines, but some abstract machines are more abstract than others. The extremes of this spectrum are characterized as follows:

- An abstract machine is an intermediate language with a small-step operational semantics [107].
- An abstract machine is a design for a real machine yet to be built.

1.2. *Related terms*

The term *abstract machine* is sometimes also used for different concepts and other terms are used for the concept of abstract machines, e.g. some authors

[*] Corresponding author. Tel.: +49-681-302-3915.
E-mail addresses: diehl@cs.uni-sb.de (S. Diehl), phh@ecs.soton.ac.uk (P. Hartel), sestoft@dina.kvl.dk (P. Sestoft)

use the terms *emulator* or *interpreter* and some use the term *virtual machine* for implementations of abstract machines, similar as we use the term *program* for implementations of an algorithm. Sun calls its abstract machine for Java the Java Virtual Machine [86,91]. The term virtual machine is widely used for the different layers of abstractions in operating systems [121] and in IBM's VM operating system virtual machines are execution environments for running several versions of the same operating system on the same machine. In theoretical computer science the term abstract machine is sometimes used for models of computation including finite state machines, Mealy machines, push down automata and Turing machines [61].

1.3. What are abstract machines used for?

In the above characterization of abstract machines their use as an intermediate language for compilation is an essential feature. As a result the implementation of a programming language consists of two stages. The implementation of the compiler and the implementation of the abstract machine. This is a typical divide-and-conquer approach. From a pedagogical point of view, this simplifies the presentation and teaching of the principles of programming language implementations. From a software engineering point of view, the introduction of layers of abstraction increases maintainability and portability and it allows for design-by-contract. Abstract machines have been successful for the design of implementations of languages that do not fit the "Von-Neumann computer" well. As a consequence most abstract machines are for exotic or novel languages. There are only few abstract machines for languages like C or Fortran. Recently abstract machines have been used for mobile code in heterogenous networks such as the Internet.

In addition to all their practical advantages abstract machines are theoretically appealing as they facilitate to prove the correctness of code generation, program analyses and transformations [20,111].

2. Where do abstract machines come from?

Abstract machines are often designed in an ad-hoc manner based on experience with other abstract machines or implementations of interpreters or compilers for the same source language. But also some systematic approaches have been investigated. Wand was one of the first to deal with the question of deriving abstract machines from the semantics of a language. In 1982, he proposed an approach based on combinators [130]. To find suitable combinators was not automated and was a difficult task, which was simplified in a later paper [131]. The CAM (1985) was derived in a similar way [34]. Another approach is based on partial evaluation of interpreters with given example programs and folding of recurring patterns in the intermediate code [44,80,98]. Finally there are approaches based on pass separation [45,56,70,89,116]. Pass separation is a transformation which splits interpreters into compiling and executing parts, the latter being the abstract machine. It has also been used in the 2BIG system (1996) to automatically generate abstract machines from programming language specifications [43,46].

3. Abstract machines for imperative programming languages

Discussions in the late fifties within the ACM and other related bodies resulted in various proposals being made for an UNCOL: A UNiversal Computer Oriented Language. Various UNCOLs have been proposed. Conway's machine [33] for example was a register machine, with two instructions. Steel's machine [119] had sophisticated adressing modes. The principle of an UNCOL is sound, but they have not been much used. We believe that this is mainly because of the lack of performance of the generated code. Chow and Ganapathi [30] give an overview of abstract machines for imperative programming languages that were current in the mid-1980s. Some believe that the Java Virtual Machine [86] of the late 1990s might finally play the role of an UNCOL, but we think that performance will remain a concern in many areas of computing.

We will now look at some successful abstract machines, which were designed for rather more modest goals:

- The Algol Object Code (1964) [109] is an abstract machine for Algol60. It has a stack, a heap and a program store. Its instructions provide mechanisms for variable and procedure scope, allocation of memory, access to variables and arrays, and call-by-value and call-by-name procedure calls.

- The P4-machine (1976) is an abstract machine for the execution of Pascal programs, developed by Wirth and colleagues [7]. The compiler from Pascal to P4 and the abstract machine code are documented in [102]. The P4 machine has fixed-length instructions. It implements block structure by a stack of activation records (frames), using dynamic and static links to implement recursion and static scoping, respectively.
- The UCSD P-machine [32] is an abstract machine for the execution of Pascal programs, with variable-length instructions. The compact bytecode of the machine has special instructions for calling Pascal's nested procedures, for calling formal procedures, for record and array indexing and index checks, for handling (Pascal) sets, for signalling and waiting on semaphores, etc. The P-machine was used in the popular UCSD Pascal system for microcomputers (ca. 1977). A commercial hardware implementation of the P-machine was made (see Section 11).
- Forth (1970) may be considered as a directly executable language of a stack-based abstract machine: expressions are written in postfix (reverse Polish notation), a subroutine simply names a code address, etc. [77,94].

4. Abstract machines for object-oriented programming languages

Abstract machines for object-oriented languages are typically stack-based and have special instructions for accessing the fields and methods of objects. Memory management is often implicit (done by a garbage collector) in these machines.

- Smalltalk-80 (1980) is a dynamically typed class-based object-oriented language, implemented by compilation into a stack-based virtual machine code. The bytecode has instructions for stack manipulation, for sending a message to an object (to access a field or invoke a method), for return, for jump, and so on [51] (the second edition [52] omits most of the material on the virtual machine).
- Self (1989) is a dynamically typed class-less object-oriented language. Self has a particularly simple and elegant stack-based virtual machine code: every instruction has a three-bit instruction op-code and a five-bit 'index', or instruction argument. The eight instructions are: push self, push literal, send message (to invoke a method or access a field), self send, super send, delegate (to a parent), return, and index extension. The bytecode is dynamically translated into efficient machine code [28,29].
- Java (1994) is a statically typed class-based object-oriented language, whose 'official' intermediate language is the statically typed Java Virtual Machine (JVM) bytecode. The JVM has special support for dynamic loading and linking, with load-time verification (including type checking) of the bytecode. The instruction set supports object creation, field access, virtual method invocation, casting an object to a given class, and so on [86]. For hardware implementations of the JVM (see Section 11).

5. Abstract machines for string processing languages

A string processing language is a programming language that focuses on string processing rather than processing numeric data. String processing languages have been around for decades in the form of command shells, programming tools, macro processors, and scripting languages. This latter category has become prominent as scripting language are used to 'glue' components together [101]. The components are typically written in a (systems) programming language, such as C, but they may be glued components themselves.

String processing languages are either implemented by interpreting a proprietary representation of the source text, or the implementation is based on some low level abstract machine. There are two reasons for using a proper abstract machine: improved execution speed and better portability. Machine independence has become less of an issue in recent years, because the number of different computer architectures has fallen dramatically over time, and because C acts as a lingua franca to virtually every platform currently in use.

We will discuss two prominent examples of early string processing languages, where an abstract machine is used mainly to achieve machine independence.

- Snobol4 [54] is a string processing language with a powerful pattern matching facility. The language has been used widely to build compilers, symbolic algebra packages, etc. The Snobol4 abstract machine (SIL) operates on data descriptors, which contain either scalar data or references, as well as the type of the data and some control information. The data representation makes it possible to treat strings as variables, and to offer data directed dispatch of operations, much in the same way as object oriented systems offer today. The machine operates a pair of stacks, a garbage collected heap (mark scan). The instruction set is designed firstly to provide efficient support for the most common operations and secondly to ease the task of porting it [53].
- ML/I [23] is a macro processor. Macro processors are based on a substitution model, whereas ordinary string processors treat strings as data to which operations are applied. Macro processors are generally more difficult to program than ordinary string processors. The ML/I macro processor is implemented via the LOWL abstract machine. This machine offers two stacks, three registers, non-recursive sub-routines and a small set of instructions. Portability has been the major driver for the design.

UNIX has had a profound influence on what we consider scripting languages today. With UNIX came the now classical tool-set comprising the shell, awk, and make. As far as we know, all of these are implemented using an internal representation close to the source text. Descendants of these tools are now appearing that use abstract machines again, mainly for speed but also for machine independence:

- Awk [1] constructs a parse tree from the source. The interpreter then traverses the parse tree, interpreting the nodes. Interior nodes correspond to an operator or control flow construct; leaves are usually pointers to data. Interpreter functions return cells that contain the computed results. Control flow interruptions like break, continue, and function return are handled specially by the main interpreter.
- Nmake [49] is a version of the make tool for UNIX, which provides a more flexible style of dependency assertions. To be able to port these new make files to older systems, Nmake can translate its input into instructions for the Make Abstract Machine (MAM). These are easy to translate into more common Makefile formats [78].
- Tcl [100] is a command language designed to be easily extensible with application specific, compiled commands. The most widely know application of Tcl is the Tk library for building Graphical User Interfaces. The flexibility of Tcl is achieved primarily by representing all data as strings and by using a simple and uniform interface to commands. For example the while construct from the Tcl language is implemented by a C procedure, taking two strings as arguments. The first string is the conditional expression and the second is the statement to be executed. The C procedure calls the Tcl command interpreter recursively to evaluate the conditional and the statements ([100], p. 321). The abstract machine does not have any stacks of its own, it relies on the C implementation.

  Since version 8.0 Tcl uses a bytecode interpreter [74].
- Perl [128] is a scripting language, with an enormous collection of modules for a wide range of applications, such as building CGI scripts for Web servers. The implementation compiles Perl code into an intermediate, tree structured representation, with each instruction pointing to the next. The abstract machine has seven stacks which are explicitly manipulated by the compiled instructions. There are six different data types, and over 300 instructions. Reference counting is used to perform storage management [118].
- Python is an object oriented scripting language [87]. Python is implemented using a stack based abstract machine. The instructions are rather like method calls, dispatching on the type of the operands found on the stack. There are over 100 instructions, organized as segments of code, with jumps to alter the flow of control. Python uses a reference count garbage collector.

  Hugunin [63] has created an implementation of JPython, which targets the Java Virtual Machine instead.

The performance of the scripting languages has above been studied by a number authors. Kernighan and van Wyk [74] compare Awk, Perl, Tcl, Java, Visual Basic, Limbo, C and Scheme. They show that depending on the benchmark and the platform, C and Java sometimes do worse than the scripting languages. Romer et al. [110], benchmark Java, Perl and Tcl using a cache level simulator of the MIPS architecture. They conclude that eventhough scripting language perform less well than C, special hardware support is not warranted.

6. Abstract machines for functional programming languages

The first abstract machines for functional languages, such as the SECD [81] and FAM [26], defined strict evaluation, also known as eager or call-by-value evaluation, in which function arguments are evaluated before the call, and exactly once. More recently, most work has focused on lazy (or call-by-need) evaluation, in which function arguments are evaluated only if needed, and at most once. One reason is that efficient implementation of strict evaluation is now well understood, so that the need to go via an abstract machine has diminished.

Central concepts in abstract machines for functional languages include:

- A *stack* in general represents the context of a nested computation. It may hold the intermediate results of pending computations, activation records of active function invocations, active exception handlers, etc. The stack is sometimes used also for storing arguments to be passed to functions.
- An *environment* maps program variables to their values.
- A *closure* is used to represent a function as a value. It typically consists of a code address (for the function body) and an environment (binding the free variables of the function body).
- A *heap* stores the data of the computation. Abstract machines usually abstract away from the details of memory management, and thus include instructions for allocating data structures in the heap, but not for freeing them; the heap is assumed to be unlimited.
- A *garbage collector* supports the illusion that the heap is unlimited; it occasionally reclaims unreachable heap space and makes it available for allocation of new objects.

6.1. Strict functional languages

- The SECD machine (1964) was designed by Landin for call-by-value evaluation of the pure lambda calculus [81]. The machine derives its name from the components of its state: an evaluation stack S, an environment E, a control C holding the instructions to execute, and a dump D holding a continuation. (i.e., a description of what must be done next).
- Cardelli's Functional Abstract Machine (1983) is a much extended and optimized SECD machine used in the first native-code implementation of ML [26,27].
- The Categorical Abstract Machine (1985) was developed by Cousineau et al. [34]. Its instructions correspond to the constructions of a Cartesian closed category: identity, composition, abstraction, application, pairing, and selection. It was the base for the CAML implementation of ML.
- The Zinc Abstract Machine (1990) developed by Leroy [82] permits more efficient execution. It is an optimized, strict version of the Krivine machine (see Section 6.2 below). This machine is the basis of the bytecode versions of Leroy's Caml Light [35,135] and Objective Caml implementations, and is used also in Moscow ML [117].

6.2. Lazy functional languages

In a lazy language, function and constructor arguments are evaluated only if needed, and then at most once. Although this can be implemented by representing an unevaluated argument by a 'thunk', a function that will evaluate the argument and replace itself with the result, efficiency calls for other approaches. An important idea due to Wadsworth is to represent the program by a graph which is rewritten by evaluation. The evaluation (rewriting) of a shared subgraph will automatically benefit all expressions referring to it. However, repeatedly searching a graph for subexpressions to rewrite is slow.

Early implementations compiled the program to a fixed set of combinators (closed lambda terms all of whose abstractions are at the head); these may be thought of as graph rewriting rules [123]. Later it was shown to be beneficial to let the program under consideration guide the choice of combinators (so-called supercombinators) [62].

- In his seminal paper [123], David Turner describes the SK-machine to support the implementation of SASL. The compiler is based on the equivalence between combinatory logic [113] and the λ-calculus [40]. It generates code for what is essentially a two instruction machine. To make the machine more efficient, Turner added further instructions, each with a functionality that is provably equivalent to a number of S and K combinators.

- The G-machine (1984) was designed by Augustsson and Johnsson for lazy (call-by-need) evaluation of functional programs in supercombinator form [10,68,104]. Instead of interpreting supercombinators as rewrite rules, they were compiled into sequential code with special instructions for graph manipulation. The G-machine is the basis of the Lazy ML [11] and HBC Haskell [13] implementations.
- The Krivine machine (1985) is a simple abstract machine for call-by-name evaluation (i.e. without sharing of argument evaluation) of the pure lambda calculus [39]. It has just three instructions, corresponding to the three constructs of the lambda calculus: variable access, abstraction, and application. A remarkable feature is that the argument stack is also the return stack (continuation).
- The Three Instruction Machine TIM (1986) is a simple abstract machine for evaluation of supercombinators, developed by Fairbairn and Wray [48]. The basic call-by-name version of this machine is quite similar to the Krivine machine. A lazy (call-by-need) version needs extra machinery to update shared function arguments; it is somewhat complicated to implement this efficiently [8].
- The Krivine machine can be made lazy just as the TIM [36,37,115]. Alternatively one may add an explicit heap and a single new instruction for making recursive let-bindings [116]. The resulting machine has been used in some theoretical studies, e.g. [112].
- The Spineless-Tagless G-machine (1989) was developed by Peyton Jones as a refinement of the G-machine [105]. It is used in the Glasgow Haskell compiler [103].

There are many more abstract machines for functional languages than we can mention here. Typically they were developed for theoretical study, or during the work on some novel language or implementation technique.

It is ultimately the performance that decides whether an abstract machine has been well designed. A comprehensive overview of over 25 functional language implementations is provided in the Pseudoknot benchmark [58].

7. Abstract machines for logic programming languages

Logic programming languages are based on predicate calculus. The program is given as a finite set of inference rules. The execution of a logic program performs logical inferences. Prolog is the most prominent logic programming language. In Prolog the rules are in a standard form known as universally quantified ‘Horn clauses’. A goal statement is used to start the computation which tries to find a proof of this goal.

Most research in compiling of Prolog programs is centered around the Warren Abstract Machine WAM (1983) which has become the de facto standard [133]. It offers special purpose instructions, which include unification instructions for various kinds of data and control flow instructions to implement backtracking. The original report by Warren [132] gives just the bare bones and there have been several efforts to present the WAM in a pedagogical way [2,50,138]. The WAM uses four memory areas: heap, stack, trail, and PDL.

- The WAM allocates structures and variables on the *heap*. Garbage collection automatically reclaims heap space allocated by structures and variables which are no longer reachable from the program.
- The *stack* contains choice points and environments. In a choice point there are entries for the address of the previous choice point, the next alternative clause (continuation pointer) and to store some of the registers of the WAM. An environment consists of the permanent variables in a clause. Conceptually the stack can be divided into two stacks, called the AND-and OR-stacks. The AND-stack contains environments and the OR-stack contains choice points.
- On the *trail*, the WAM keeps track of which bindings have to be retracted after a clause fails and before an alternative clause can be tried, i.e. during backtracking.
- Finally the *push down list, PDL,* contains pairs of nodes, which have to be considered next by the unification algorithm. Unification matches the current goal with the head of a clause and binds variables in both the goal and the head.

Research has focussed on the generation of optimized WAM code and resulting extensions and modifications of the WAM have been proposed. Some of the techniques, which have been investigated, are indexing of clauses, environment trimming, register allocation [66,88] and tabling of goals [108]. Data flow analysis, in particular abstract interpretation, and tail recursion optimizations have been the basis of efficient implementations of Prolog [125,127].

8. Abstract machines for hybrid programming languages

Programming language researchers try to combine the best of different language paradigms in hybrid programming languages. For functional logic languages abstract machines have been designed as extensions of abstract machines for functional languages [79,97] or as extensions of the WAM [21,57]. The WAM has also been the basis for abstract machines for constraint logic programming languages [16,65], the concurrent, constraint logic programming language OZ [90] and the concurrent, real time language Erlang [9].

9. Abstract machines for parallel programming languages

As noted by Blair [18], parallel and distributed models converge due to trends towards high-speed networks, platform independence and micro-kernel based operating systems. Several such models are discussed in [73], most notably the Parallel Virtual Machine PVM (1990) which serves as an abstraction to program sets of heterogeneous computers as a single computational resource [15,120]. The Threaded Abstract Machine TAM (1993) [38] and a similar, but simpler abstract machine [3,47] have been proposed as general target architectures for multi-threading on highly parallel machines.

Parallel and distributed architectures provide computation power which programming language implementations on these systems try to exploit.

Pure functional languages are referentially transparent, and parallel evaluation of e.g. the arguments of a function invocation would seem a promising idea. Indeed, several abstract machines have been suggested which implement parallel graph reduction, e.g. $\langle \nu, G \rangle$-machine [12], GRIP [106], GUM [122], and DREAM [22]. Also an abstract machine for parallel proof systems [67] is based on parallel graph reduction. A critical review of parallel implementations of functional languages, in particular of lazy languages, is given by Wilhelm et al. [137]. They observe that the exploitation of natural parallelism in functional programming languages has not been successful so far [137]. In general, giving the programmer control over the parallelism in the language allows for better results, e.g. the PCKS-machine [96].

Parallelism naturally arises in logic programming: Several clauses for the same goal (AND-parallelism) or all goals in a clause (OR-parallelism) can be tried in parallel. AND-Parallel models have been proposed in [59,85], some of the OR-Parallel models are the SRI model [134], the Argonne model [24], the BC machine [4], the MUSE model [5,6] and the model proposed in [31]. Furthermore there have been attempts to combine AND and OR parallelism [55]. There have also been parallel abstract machines, which are totally different from the WAM. One of these is the PPAM [71], which is based on a dataflow model.

Some of the important issues in implementing parallel abstract machines for programming languages are static and dynamic scheduling [19,114], granularity of tasks, distributed garbage collection [69] and code and thread migration [126,136].

10. Special-purpose abstract machines

Abstract machines are not only used for translation of programming languages, but also as intermediate levels of abstraction for other purposes. Term rewriting [42] is a model of computation used in various areas of computer science, including symbolic computation, automated theorem proving and execution of algebraic specifications. Abstract machines for term rewriting systems include the abstract rewriting machine ARM [72], μARM [129] and TRAM [99].

Portability is the main reason for the success of DVI [76] and PostScript [64] as page-description languages. DVI is a simple language without control-flow constructs, whereas PostScript is a full programming language in the tradition of Forth. Both are used as intermediate languages by text processing systems. They are either further compiled into the language of a certain printer or interpreted by the printer's built-in processor.

In natural language parsing abstract machines based on the WAM are investigated. In contrast to usual Prolog programs, terms in unification grammars tend to be large and thus efficient unification instructions are added to the WAM [17,139].

The Hypertext Abstract Machine (1988) is a server for a hypertext storage system [25]. The data structures of the machine are graphs, contexts, nodes, links and attributes. The instructions of the machine initiate transactions with the server to access and modify these.

11. Concrete abstract machines

A computer processor (CPU) could be considered a concrete hardware realization of an abstract machine, namely the processor's design. While this view is rather extreme when applied to processors such as the x86, SPARC, MIPS, or HP-PA, it makes more sense when applied to unconventional, special-purpose processors or abstract machines.

For many years (roughly, from the early 1970s to the late 1980s) it was believed that efficient implementation of symbolic languages, such as functional and logic languages, would require special-purpose hardware. Several such hardware implementations were undertaken, and some resulted in commercial products. However, the rapid development of conventional computer hardware, and advances in compiler and program analysis technology, nullified the advantages of special-purpose hardware, even microcoded implementations. Special-purpose hardware was too expensive to build and could not compete with stock hardware.

A tell-tale sign is that the conference series *Functional Programming and Computer Architecture* (initiated in 1981) published few papers on concrete computer architecture, and when merging with the Lisp conference series in 1996, dropped 'Computer Architecture' from its title.

Some examples of concrete hardware realizations of abstract machines are:

- The Burroughs B5000 processor (1961) had hardware support for efficient stack manipulation, such as keeping the top few elements of the stack in special CPU registers. The goal was to support the implementation of Algol 60 and other block-structured languages. Subsequently many machines with hardware stack support have been developed (see [77]).
- A Lisp machine project was initiated at MIT in 1974 and led to the creation of the company Symbolics in 1980. Symbolics Lisp Machines had a special-purpose processor and memory, with support for e.g. the run-time type tags required by Lisp. The entire operating system and development environment were written in Lisp. By 1985 the company had sold 1500 Lisp Machines; by 1996 it was bankrupt.
- The Pascal Microengine Computer (1979) is a hardware implementation of the UCSD P-code abstract machine [92](see Section 3). Analogously to the Lisp machines, the operating system and development environment were written entirely in Pascal. The machine was commercially available in the early 1980s.
- ALICE (Applicative Language Idealized Computing Engine) [41] by Darlington and Reeve was the first hardware implementation of a reduction machine. It was built using 40 transputers connected by a multi-stage switching network.
- Kieburtz and others (1985) designed a hardware implementation of Augustsson and Johnsson's G-machine (see Section 6.2) for graph reduction of lazy functional languages [75]. Simulations of the processor and memory management system were done, but the hardware was never built.
- The Norma was created by the Burroughs company (1986) as a research processor for high speed graph reduction in functional programming languages (see e.g. [77]).
- Scheme-81 is a chip implementing an evaluator (abstract machine) for executing Scheme [14].
- A number of special-purpose machines for Prolog execution have been developed, mostly based on the WAM and modifications thereof. Several machines were designed within the Japanese Fifth Generation Project (1982–1992), and a total of around 800 such machines were built; they were used mostly inside the project. Other hardware implementations include the KCM project (1988) at ECRC in Europe, and the VLSI-BAM, an implementation of the Berkeley Abstract Machine, designed at Berkeley in the USA. For more information and references (see [124], Section 3.2).

- The Transputer (1984) [84] is a special-purpose microprocessor for the execution of Occam [83], a parallel programming language with synchronous communication, closely based on Hoare's theoretical language CSP [60]. It has special hardware and instructions for creating networks of Transputers.
- MuP21 (1993) is an extremely simple but fast microprocessor for executing Forth programs. It has two stacks, 20-bit words, just 24 (five-bit) instructions, and its implementation requires only 7000 transistors [95].
- A series of Java microprocessors which directly execute Java Virtual Machine bytecode (see Section 4) and support also conventional imperative languages was announced by Sun Microsystems in 1996. Technical specifications for the microJava 701 processor were available by early 1998 [93], but apparently the chip was not yet in volume production by early 1999.

12. Conclusion

For almost 40 years abstract machines have been used for programming language implementation. As new languages appear, so will abstract machines as tools to handle the complexity of implementing these languages. While abstract machines are a useful tool to bridge the gap between a high level language and a low level architecture, much work remains to be done to develop a theory of abstract machines. Such a theory is necessary to support the systematic development of abstract machines from language and architecture specifications.

Acknowledgements

We thank David Barron, Hugh Glaser, Brian Kernighan, John Ousterhout, Guido van Rossum and Reinhard Wilhelm for their help with our research.

References

[1] A.V. Aho, B.W. Kernighan, P.J. Weinberger, The AWK Programming Language, Addison-Wesley, Reading, MA, 1988.
[2] H. Aït-Kaci, Warren's Abstract Machine – A Tutorial Reconstruction, MIT Press, Cambridge, MA, 1991.
[3] Engelhardt, Alexander, Wendelborn, An overview of the Adl language project, Proceedings of the Conference on High Performance Functional Computing, Denver, Colorado, 1995.
[4] K.A.M. Ali, OR-parallel execution of Prolog on BC-machine, Proceedings of the Fifth International Conference and Symposium on Logic Programming, MIT Press, Cambridge, MA, 1988, pp. 1531–1545.
[5] K.A.M. Ali, R. Karlsson, Scheduling OR-Parallelism in MUSE, Proceedings of the 1990 North American Conference on Logic Programming, MIT Press, Cambridge, MA, 1990, pp. 807–821.
[6] K.A.M. Ali, R. Karlsson, The MUSE Or-Parallel Prolog Model and its Performance, Proceedings of the Eighth International Conference on Logic Programming, MIT Press, Cambridge, MA, 1991, pp. 757–776.
[7] U. Ammann, Code Generation of a Pascal-Compiler, In: D.W. Barron (Ed.), Pascal – The Language and its Implementation, Wiley, New York, 1981.
[8] G. Argo, Improving the three instruction machine, Fourth International Conference on Functional Programming Languages and Computer Architecture, Imperial College, London, Addison-Wesley, Reading, MA, September 1989, pp. 100–112.
[9] J.L. Armstrong, B.O. Däcker, S.R. Virding, M.C. Williams, Implementing a functional language for highly parallel real time applications, Proceedings of the Conference on Software Engineering for Telecommunication Systems and Services (SETSS'92), Florence, 1992.
[10] L. Augustsson, A compiler for lazy ML, 1984 ACM Symposium on Lisp and Functional Programming, Austin, Texas, ACM, 1984, pp. 218–227.
[11] L. Augustsson, The interactive lazy ML system, J. Funct. Programming 3 (1) (1993) 77–92.
[12] L. Augustsson, T. Johnsson, Parallel graph reduction with the $\langle \nu, G \rangle$ machine, Proceedings of the Conference on Functional Programming Languages and Computer Architecture (FPCA'89), London, ACM, New York, 1989, pp. 202–213.
[13] L. Augustsson, HBC – The Chalmers Haskell compiler, Web Page, 1999, URL: http://www.cs.chalmers.se/augustss/hbc.html.
[14] J. Batali et al., The Scheme-81 architecture, system and chip, MIT Conference on Advanced Research in VLSI, 1982.
[15] A. Beguelin, J. Dongarra, A. Geist, B. Manchek, V. Sunderam, Recent Enhancements to PVM, Internat. J. Supercomput. Appl. 9 (2) (1995).
[16] C. Beierle, Formal design of an abstract machine for constraint logic programming, IFIP Congress, vol. 1, 1994, pp. 377–382.
[17] C. Beierle, G. Meyer, H. Semele, Extending the Warren Abstract Machine to Polymorphic Order-sorted Resolution, Technical Report IWBS Report 181, Institute for Knowledge-based Systems, Stuttgart, 1991.
[18] G.S. Blair, A convergence of parallel and distributed computing, in: M. Kara, J.R. Davy, D. Goodeve, J. Nash (Eds.), Abstract Machine Model for Parallel and Distributed Computing, IOS Press, Amsterdam, 1996.

[19] J. Blazwicz, K.H. Ecker, G. Schmidt, J. Weglarz, Scheduling in Computer and Manufacturing Systems, Springer, Berlin, 1994.

[20] E. Börger, D. Rosenzweig, The WAM – Definition and Compiler Correctness, in: C. Beierle, L. Plümer (Eds.), Logic Programming: Formal Methods and Practical Applications, North-Holland, Amsterdam, 1995, pp. 22–90.

[21] P.G. Bosco, C. Cecchi, C. Moiso, An extension of WAM for K-LEAF: a WAM-based compilation of conditional narrowing, Proceedings of the Sixth International Conference on Logic Programming (Lisboa), MIT Press, Cambridge, MA, 1989.

[22] S. Breitinger, U. Klusik, R. Loogen, Y. Ortega-Mallén, R. Pena, DREAM: The DistRibuted Eden Abstract Machine, in: Implementation of Functional Languages – Selected Papers of the Ninth International Workshop (IFL'97), Lecture Notes in Computer Science, vol. 1467, Springer, Berlin, 1997, pp. 250–269.

[23] P.J. Brown, Macro Processors and Techniques for Portable Software, Wiley, Chichester, UK, 1988.

[24] R. Butler, T. Disz, E. Lusk, R. Olson, R. Overbeek, R. Stevens, Scheduling Or-Parallelism: An Argonne Perspective, in: Proceedings of the Fifth International Conference and Symposium on Logic Programming, MIT Press, Cambridge, MA, 1988.

[25] B. Campbell, J.M. Goodman, HAM: a general purpose hypertext abstract machine, Commun. ACM 31 (7) (1988) 856–861.

[26] L. Cardelli, The functional abstract machine, Technical Report TR-107, AT&T Bell Labs, 1983.

[27] L. Cardelli, Compiling a functional language, Lisp and Functional Programming, ACM, New York, 1984.

[28] C. Chambers, D. Ungar, E. Lee, An efficient implementation of Self, a dynamically typed object-oriented language based on prototypes, in: OOPSLA'89, New Orleans, LA, October 1989, SIGPLAN Notices 24 (10) (1989) 49-70.

[29] C. Chambers, D. Ungar, E. Lee, An efficient implementation of Self, a dynamically typed object-oriented language based on prototypes, Lisp Symbol. Comput. 4 (3) (1991) 57–95.

[30] F.C. Chow, M. Ganapathi, Intermediate languages in compiler Construction-A bibliography, ACM SIGPLAN Notices 18 (11) (1983) 21–23.

[31] A. Ciepielewski, S. Haridi, B. Hausman, Or-Parallel Prolog on Shared Memory Multiprocessors, J. Logic Programming 7 (1989) 125–147.

[32] R. Clark, S. Koehler, The UCSD Pascal Handbook, Prentice-Hall, Englewood Cliffs, NJ, 1982.

[33] M.E. Conway, Proposal for an UNCOL, Commun. ACM 1 (10) (1958) 5–8.

[34] G. Cousineau, P.-L. Curien, M. Mauny, The Categorial Abstract Machine, Proceedings of FPCA'85, Springer, LNCS 201, 1985.

[35] G. Cousineau, M. Mauny, The Functional Approach to Programming, Cambridge University Press, Cambridge, 1998.

[36] P. Crégut, An abstract machine for the normalization of λ-terms, ACM Conference on Lisp and Functional Programming, Nice, France, ACM Press, 1990, pp. 333–340.

[37] P. Crégut, Machinesá environnement pour la réduction symbolique et l'évaluation partielle, Ph.D. Thesis, Université Paris VII, France, 1991.

[38] D. Culler, S.C. Goldstein, K. Schauser, T. von Eicken, TAM – A Compiler Controlled Threaded Abstract Machine, Special Issue on Dataflow, J. Parallel Distrib. Comput. 1993.

[39] P.L. Curien, The $\lambda\rho$-calculus: an abstract framework for environment machines, Rapport de Recherche LIENS-88-10, Ecole Normale Supérieure, Paris, France, 1988.

[40] H.B. Curry, R. Feys, Combinatory Logic, vol. I, North-Holland, Amsterdam, 1958.

[41] J. Darlington, M.J. Reeve, ALICE: A multiple-processor reduction machine for the parallel evaluation of applicative languages, Proceedings of FPCA'81, 1981, pp. 65–76.

[42] N. Dershowitz, J.-P. Jouannaud, Rewrite systems, in: J. van Leeuwen (Ed.), Handbook of Theoretical Computer Science, vol. B, North-Holland, Amsterdam, 1990, pp. 243–320.

[43] S. Diehl, Semantics-directed generation of compilers and abstract machines, Ph.D. Thesis, University Saarbrücken, Germany 1996, http://www.cs.uni-sb.de/~diehl/phd.html.

[44] S. Diehl, An experiment in abstract machine design, Software – Practice and Experience 27 (1) (1997).

[45] S. Diehl, Transformations of evolving algebras, Proceedings of the VIII International Conference on Logic and Computer Science LIRA'97, Novi Sad,Yugoslavia, 1997, pp. 43–50.

[46] S. Diehl, Natural semantics-directed generation of compilers and abstract machines, Formal Aspects of Computing, in press.

[47] Engelhardt, Wendelborn, A partitioning-independent paradigm for nested data parallelism, Internat. J. Parallel Programming 24 (2) (1996).

[48] J. Fairbairn, S.C. Wray, TIM: A simple, lazy abstract machine to execute supercombinators, in: G. Kahn (Ed.), Functional Programming Languages and Computer Architecture, Portland, Oregon, Lecture Notes in Computer Science, vol. 274, Springer, Berlin, 1987, pp. 34–45.

[49] G. Fowler, A case for make, Software–practice and experience 20 (S1) (1990) 35–46.

[50] J. Gabriel, T. Lindholm, E.L. Lusk, R.A. Overbeek, A Tutorial on the Warren Abstract Machine, Technical Report ANL-84-84, Argonne National Laboratory, Argonne, IL, 1985.

[51] A. Goldberg, D. Robson, Smalltalk-80, The Language and Its Implementation, Addison-Wesley, Reading, MA, 1983.

[52] A. Goldberg, D. Robson, Smalltalk-80, The Language and Its Implementation, Addison-Wesley, Reading, MA, 1989.

[53] R.E. Griswold, The Macro Implementation of SNOLBOL4, Freeman, New York, 1972.

[54] R.E. Griswold, J.F. Poage, I.P. Polonsky, The SNOLBOL4 programming language, Prentice-Hall, Englewood Cliffs, New Jersey, 1971.

[55] G. Gupta, B. Jayaraman, A model for AND-OR parallel execution of logic programs, Proceedings of the 18th International Conference on Parallel Processing, 1989.

[56] J. Hannan, Operational semantics-directed compilers and machine architectures, ACM Transactions on Programming Languages and Systems 16 (4) (1994) 1215–1247.

[57] M. Hanus, Compiling logic programs with equality, Proceedings of Second International Workshop on Programming Language Implementation and Logic Programming (PLILP'90), Lecture Notes in Computer Science, vol. 456, Springer, Berlin, 1990, pp. 387–401.

[58] P.H. Hartel, M. Feeley, M. Alt, L. Augustsson, P. Baumann, M. Beemster, E. Chailloux, C.H. Flood, W. Grieskamp, J.H.G, van Groningen, K. Hammond, B. Hausman, M.Y. Ivory, R.E. Jones, J. Kamperman, P. Lee, X. Leroy, R.D. Lins, S. Loosemore, N. Röjemo, M. Serrano, J.-P. Talpin, J. Thackray, S. Thomas, P. Walters, P. Weis, P. Wentworth, Benchmarking implementations of functional languages with "pseudoknot", a Float-Intensive benchmark, J. Funct. Programming 6 (4) (1996) 621–655, ftp.wins.uva.nl pub computer-systems functional reports JFP pseudoknotI.ps.Z.

[59] M.V. Hermenegildo, An abstract machine based execution model for parallel execution of logic programs, Ph.D. Thesis, University of Texas at Austin, Austin, 1986.

[60] C.A.R. Hoare, Communicating Sequential Processes, Prentice-Hall, Englewood Cliffs, NJ, 1985.

[61] J.E. Hopcroft, J.D. Ullman, Introduction to Automata Theory, Languages, and Computation, Addison-Wesley, Reading, MA, 1979.

[62] R.J.M. Hughes, Super-combinators: A new implementation method for applicative languages, ACM Symposium on Lisp and Functional Programming, Pittsburgh, Pennsylvania, ACM, August 1982, pp. 1–10.

[63] J. Hugunin, Python and Java – the best of both worlds, Proceedings of the Sixth International Python Conference, San Jose, California, October 1997, Corporation for National Research Initiatives, Reston, Virginia.

[64] Adobe Systems, Inc., PostScript Language Reference Manual, Addison-Wesley, Reading, MA, 1990.

[65] J. Jaffar, P.J. Stuckey, S. Michaylov, R.H.C. Yap, An Abstract Machine for CLP(R), in: PLDI'92, San Francisco, SIGPLAN Notices, 1992.

[66] G. Janssens, B. Demoen, A. Mariën, Improving the register allocation in WAM by reordering unificiation, Proceedings of the Fifth International Conference and Symposium on Logic Programming, MIT Press, Cambridge, MA, 1988, pp. 1388–1402.

[67] R. Johnson, K. Shen, M. Fisher, J. Keane, A. Nisbet, An abstract machine for prototyping parallel proof mechanisms, in: M. Kara, J.R. Davy, D. Goodeve, J. Nash (Eds.), Abstract Machine Model for Parallel and Distributed Computing, IOS Press, Amsterdam, 1996.

[68] T. Johnsson, Efficient compilation of lazy evaluation, Proceedings of the ACM SIGPLAN'84 Symposium on Compiler Construction, SIGPLAN Notices 19 (6) (1984) 58–69.

[69] R. Jones, R. Lins, Garbage Collection – Algorithms for Automatic Dynamic Memory Management, Wiley, New York, 1996.

[70] U. Jørring, W.L. Scherlis, Compilers and staging transformations, Proceedings of the 13th ACM Symposium on Principles of Programming Languages, 1986.

[71] P. Kacsuk, Execution Models of Prolog for Parallel Computers, MIT Press, Cambridge, MA, 1990.

[72] J.F.T. Kamperman, H.R. Walters, ARM – Abstract rewriting machine, in: H.A. Wijshoff (Ed.), Computing Science in the Netherlands, 1993.

[73] M. Kara, J.R. Davy, D. Goodeve, J. Nash (Eds.), Abstract Machine Model for Parallel and Distributed Computing, IOS Press, Amsterdam, 1996.

[74] B.W. Kernighan, C.J. Van Wyk, Timing trials, or the trials of timing: experiments with scripting and user-interface languages, Software – practice and experience 28 (8) (1998) 819–843.

[75] R.B. Kieburtz, The G-machine: A fast, graph-reduction evaluator, in: J.-P. Jouannaud (Ed.), Function Programming Languages and Computer Architecture, Nancy, France, September 1985, Lecture Notes in Computer Science, vol. 201. Springer, Berlin, 1985.

[76] D.E. Knuth, TeX: The Program, Addison-Wesley, Reading, MA, 1986.

[77] P. Koopman, Stack Computers: the new wave, Ellis Horwood, Chichester, UK, 1989, URL: http://www.cs.cmu.edu/koopman/stack/computers/.

[78] D.G. Korn, KSH – an extensible high level language, Very High Level Languages (VHLL), Santa Fe, New Mexico, Usenix Association, Berkely, CA, 1994, pp. 129–146.

[79] H. Kuchen, R. Loogen, J.J. Moreno-Navarro, M. Rodríguez-Artalejo, Graph-based implementation of a functional logic language, Proceedings of European Symposium on Programming Languages (ESOP'90), Lecture Notes in Computer Science, vol. 432, Springer, Berlin, 1990, pp. 271–290.

[80] P. Kursawe, How to invent a Prolog machine, Proceedings of the Third International Conference on Logic Programming, Lecture Notes in Computer Science, vol. 225, Springer, Berlin, 1986, pp. 134–148.

[81] P.J. Landin, The mechanical evaluation of expressions, Comput. J. 6 (4) (1964).

[82] X. Leroy, The Zinc experiment: An economical implementation of the ML language, Rapport Technique 117, INRIA Rocquencourt, France 1990.

[83] INMOS Limited, Occam 2 Reference Manual, Prentice-Hall, London, 1988, ISBN 0-13-629312-3.

[84] INMOS Limited, Transputer Instruction Set – A Compiler Writer's Guide, Prentice-Hall, London, 1988, ISBN 0-13-929100-8.

[85] Y.-J. Lin, V. Kumar, AND-parallel execution of logic programs on a shared memory multiprocessor: a summary of results, Proceedings of the Fifth International Conference and Symposium on Logic Programming, MIT Press, Cambridge, MA, 1988, pp. 1123–1141.

[86] T. Lindholm, F. Yellin, The Java Virtual Machine Specification, Addison-Wesley, Reading, MA, 1996.

[87] M. Lutz, Programming Python, O'Reilly, Sebastopol, California, October 1996.

[88] L. Matyska, A. Jergova, D. Toman, Register allocation, in WAM, Proceedings of the Eighth International Conference on Logic Programming, MIT Press, Cambridge, MA, 1991.

[89] S. McKeever, A framework for generating compilers from natural semantics specifications, in: P.D. Mosses (Ed.),

Proceedings of the First Workshop on Action Semantics, BRICS-NS-94-1, University of Aarhus, Denmark, 1994.

[90] M. Mehl, R. Scheidhauer, C. Schulte, An abstract machine for Oz, in: M. Hermenegildo, S.D. Swierstra (Eds.), Proceedings of the Seventh International Symposium, PLILP'95, vol. LNCS 982, Springer, Berlin, 1995.

[91] J. Meyer, T. Downing, Java Virtual Machine, O'Reilly, Sebastopol, California, 1997.

[92] The Microengine Company, Newport Beach, California, USA, Pascal Microengine Computer User's Manual, 1979.

[93] Sun Microsystems, Microjava-701 java processor, Sun Microsystems Data Sheet, January 1998, URL: http://www.sun.com/microelectronics/microJava-701.

[94] C. Moore, Forth: A new way to program a mini-computer, Astron. Astrophys. (Suppl.) 15 (1974) 497-511.

[95] C. Moore, C.H. Ting, MuP21 – a high performance MISC processor, Forth Dimensions, January 1995. URL: http://www.UltraTechnology, com/mup21.html.

[96] L. Moreau, The PCKS-machine: An abstract machine for sound evaluation of parallel functional programs with first-class continuations, Proceedings of the European Symposium on Programming Languages (ESOP'94), Lecture Notes in Computer Science, vol. 788, Springer, Berlin, 1994, pp. 424–438.

[97] A. Mück, Camel: An extension of the categorial abstract machine to compile functional logic programs, PLILP'92, Lecture Notes in Computer Science, vol. 631, Springer, Berlin, 1992.

[98] U. Nilsson, Towards a methodology for the design of abstract machines for logic programming, J. Logic Programming 16 (1993) 163–188.

[99] K. Ogata, K. Ohhara, K. Futatsugi, TRAM: An abstract machine for order-sorted conditional term rewriting systems, Proceedings of the Eighth International Conference on Rewriting Techniques and Applications (RTA'97), Lecture Notes in Computer Science, vol. 1232, Springer, Berlin, 1997, pp. 335–338.

[100] J.K. Ousterhout, Tcl and the Tk tookit, Addison-Wesley, Reading, MA, 1994.

[101] J.K. Ousterhout, Scripting: Higher level programming for the 21st century, IEEE Computer 31 (3) (1998) 23–30.

[102] S. Pemberton, M. Daniels, Pascal Implementation: The P4 Compiler and Interpreter, Ellis Horwood, Chichester, UK, 1983, ISBN 0-13-653-0311.

[103] S.L. Peyton Jones, C.V. Hall, K. Hammond, W.D. Partain, P.L. Wadler, The Glasgow Haskell compiler: a technical overview, Joint Framework for Information Technology (JFIT) Technical Conference, Keele, England, March 1993, DTI/SERC, pp. 249–257.

[104] S.L. Peyton Jones, The Implementation of Functional Programming Languages, Prentice-Hall, Englewood Cliffs, NJ, 1987.

[105] S.L. Peyton Jones, Implementing lazy functional languages on stock hardware, the spineless tagless G-machine, J. Funct. Programming 2 (2) (1992) 127–202.

[106] S.L. Peyton Jones, C. Clark, J. Salkild, M. Hardie, GRIP: a high-performance architecture for parallel graph reduction, in: T.J. Fountain, M.J. Shute (Eds.), Multi-Processor Computer Architectures, North-Holland, Amsterdam, 1990.

[107] G.D. Plotkin, A structural approach to operational semantics, Technical Report FN-19, DAIMI, Aarhus University, Denmark, 1981.

[108] I.V. Ramakrishnan, P. Rao, K. Sagonas, T. Swift, D.S. Warren, Efficient tabling mechanisms for logic programs, in: L. Sterling (Ed.), Proceedings of the 12th International Conference on Logic Programming ,Tokyo, MIT Press, Cambridge, MA, 1995.

[109] B. Randell, L.J. Russel, Algol60 Implementation, Academic Press, New York, 1964.

[110] T.H. Romer, D. Lee, G.M. Voelker, A. Wolman, W.A. Wong, J.-L. Baer, B.N. Bershad, H.M. Levy, The structure and performance of interpreters, Proceedings of the 7th Int. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS), Cambridge, MA, September 1996, ACM, New York, 1996, pp. 150–159.

[111] D.M. Russinoff, A verified Prolog compiler for the Warren abstract machine, J. Logic Programming 13 (1992) 367–412.

[112] P.M. Sansom, S.L. Peyton Jones, Formally based profiling for higher-order functional languages, ACM Trans. Programming Languages Systems 19 (2) (1997) 334–385.

[113] M. Schönfinkel, Über die Bausteine der mathematischen Logik, Mathematische Annalen 92 (6) (1924) 305–316.

[114] H. Seidl, R. Wilhelm, Probabilistic load balancing for parallel graph reduction, Proceedings IEEE Region 10 Conference, IEEE Press, New York, 1989.

[115] P. Sestoft, Analysis and efficient implementation of functional programs, Ph.D. Thesis, DIKU, University of Copenhagen, Denmark, 1991, DIKU Research Report 92/6.

[116] P. Sestoft, Deriving a lazy abstract machine, J. Programming 7 (3) (1997).

[117] P. Sestoft, Moscow, ML, Web Page, 1999, URL: http://www.dina.kvl.dk/ sestoft/mosml.html.

[118] S. Srinivasan, Advanced Perl Programming, O'Reilly, Sebastopol, California, August 1997.

[119] T.B. Steel Jr., A first version of UNCOL, Western Joint Comp. Conf., pp. 371–378. IRE and AIEE and ACM and AFIPS, New York, May 1961.

[120] V.S. Sunderam, PVM: a framework for parallel distributed computing, Software – Practice and Experience 2 (4) (1990) 315–339.

[121] A.S. Tanenbaum, Modern Operating Systems, Prentice-Hall, Englewood Cliffs, NJ, 1992.

[122] P.W. Trinder, K. Hammond, J.S. Mattson Jr., A.S. Patridge, GUM: a portable parallel implementation of Haskell, Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'96), Philadelphia, 1996.

[123] D.A. Turner, A new implementation technique for applicative languages, Software – Practice and Experience 9 (1979) 31–49.

[124] P. van Roy, 1983–1993: The wonder years of sequential Prolog implementation, J. Logic Programming (1994).

[125] P. van Roy, A.M. Despain, The benefits of global dataflow analysis for an optimizing Prolog compiler, Proceedings of

the 1990 North American Conference on Logic Programming, MIT Press, Cambridge, MA, 1990.
[126] P. van Roy, S. Haridi, P. Brand, G. Smolka, M. Mehl, R. Scheidhauer, Mobile objects in distributed Oz, ACM Trans. Programming Languages Systems 19 (5) (1997) 804–851.
[127] P.L. van Roy, Can logic programming execute as fast as imperative programming, Ph.D. Thesis, University of California at Berkley, Berkley, 1990.
[128] L. Wall, T. Christiansen, R.L. Schwartz, Programming Perl, 2nd ed., O'Reilly, Sebastopol, California, September 1996.
[129] H.R. Walters, J.F.Th. Kamperman, EPIC: an equational language – abstract machine and supporting tools, Proceedings of the Seventh Conference on Rewriting Techniques and Applications, Lecture Notes in Computer Science, vol. 1103, Springer, Berlin, 1996, pp. 424–427.
[130] M. Wand, Semantics-directed machine architecture, Proceedings of POPL 82, 1982.
[131] M. Wand, From interpreter to compiler: a representational derivation, in: H. Ganzinger, N.D. Jones (Eds.), Programs as Data Objects, Lecture Notes in Computer Science, vol. 217, Springer, Berlin, 1986.
[132] D.H.D. Warren, Implementing Prolog – compiling predicate logic programs, D.A.I Research Report No. 40, Edinburgh, 1977.
[133] D.H.D. Warren, An abstract Prolog instruction set, Technical Note 309, SRI International, Menlo Park, CA, 1983.
[134] D.H.D. Warren, The SRI-Model for OR-Parallel Execution of Prolog – Abstract Design and Implementation Issues, IEEE Symposium on Logic Programming, San Francisco, 1987, pp. 125–133.
[135] Pierre Weis, The Caml Language, Web Page, 1999, URL: http://pauillac.inria.fr/caml/.
[136] B. Weissman, B. Gomes, J.W. Quittek, M. Holtkamp, Efficient fine-grain thread migration with active threads, Proceedings of 12th International Parallel Processing Symposium and 9th Symposium on Parallel and Distributed Processing, Orlando, Florida, 1996.
[137] R. Wilhelm, M. Alt, F. Martin, M. Raber, Parallel implementation of functional languages, in: M. Dam (Ed.), Fifth LOMAPS Workshop, Analysis and Verification of Multiple-Agent Languages, Lecture Notes in Computer Science, vol. 1192, Springer, Berlin, June 1997.
[138] R. Wilhelm, D. Maurer, Compiler Design: Theory, Construction, Generation, Addison-Wesley, Reading, MA, 1995.
[139] S. Wintner, E. Gabrilovich, N. Francez, Amalia —a unified platform for parsing and generation, Proceedings of Recent Advances in Natural Language Processing, Tzigov Chark, Bulgaria, 1997.

Stephan Diehl received his M.S. in computer science as a Fulbright scholar at Worcester Polytechnic Institute, Massachusetts, in 1993, and his Ph.D. as a DFG scholar at Saarland University, Germany, in 1996. He is currently assistant professor at Saarland University and works in the research group of Prof. Reinhard Wilhelm. Stephan Diehl is the author of two books with Addison-Wesley and in the last three years he has published over 20 scientific papers about programming language theory, internet technology, visualization, Java and VRML. He teaches courses and seminars at the university and in the industry about these topics. Since summer 1998 he is project leader of the project GANIMAL sponsored by DFG (German Research Council).

Dr. Pieter Hartel received his Ph.D. degree in Computer Science from the University of Amsterdam in 1989. He has worked at CERN in Geneva and the Universities of Nijmegen and Amsterdam. He is currently a Senior Lecturer at the University of Southampton. Dr. Hartel has consulted for IT companies in the USA and in Europe. He has written a text book on programming, and over 75 publications in the areas of computer architecture, programming language design and formal methods. He is secretary and founding member of IFIP working group 8.8 on smart cards, and he is general secretary of the European Association for Programming Languages and Systems (EAPLS).

Peter Sestoft received his Ph.D. in computer science from DIKU at the University of Copenhagen in 1991. He has worked at the Technical University of Denmark and spent a sabbatical at AT&T Bell Labs in New Jersey. Currently he is associate professor in computer science at the Royal Veterinary and Agricultural University and a part-time associate of the new IT University in Copenhagen. Peter Sestoft co-pioneered self-applicable partial evaluation (with Jones and Sondergaard), and co-authored a book on partial evaluation (with Jones and Gomard). He has developed program analyses, transformations and abstract machines for functional languages. With Sergei Romanenko he developed and now maintains the Moscow ML implementation.