ELSEVIER

Future Generation Computer Systems 16 (2000) 831–839

FGCS
FUTURE GENERATION COMPUTER SYSTEMS

Visualizing principles of abstract machines by generating interactive animations

Stephan Diehl*, Thomas Kunze

FB-14 Informatik, Universität des Saarlandes, Postfach 15 11 50, 66041 Saarbrücken, Germany

Accepted 24 May 1999

Abstract

In this paper we describe the design rationale of GANIMAM, a web-based system which generates interactive animations of abstract machines from specifications. Common principles of abstract machines come into play at three levels: the design of the specification language, the choice of graphical annotations to visualize higher-level abstractions and the use of the system to explore and better understand known and detect new principles.

Keywords: Abstract machines; Software visualization; Animation

1. Introduction

In the GANIMAL project we develop learning software for compiler design. Conceptually the computations performed by a compiler can be divided into several phases. For most of these phases there exist specification languages to define such a phase and generators which given the specification generate an implementation of the phase (e.g. LEX for lexical analysis, YACC for syntax analysis and PAG for semantical analysis [2]). As a part of our project we develop generators, which do not only generate implementations, but also visualizations of the compiler phase from a standard specification. In this paper we describe the design rationale of GANIMAM, our web-based generator for interactive animations of abstract machines. Fig. 1. shows a snapshot of such an animation. GANIMAM was designed to help students to learn about and experiment with abstract machines.

Abstract machines provide intermediate target languages for compilation. First the compiler generates code for the abstract machine, then this code can be interpreted or further compiled into real machine code. By dividing compilation into two stages, abstract machines increase portability and maintainability of compilers. The instructions of an abstract machine are tailored to specific operations required to implement operations of a source language or even better for languages of the same language paradigm.

In the following sections we describe how to use GANIMAM and what is generated by the system. Then we discuss the design of the specification language. Next we explain how we enhance animations by introducing annotations. Finally we discuss the benefits of using GANIMAM and its generated interactive animations both as a development tool and as a part of a learning software.

* Corresponding author. Tel.: +49-681-302-3915.
E-mail address: diehl@cs.uni-sb.de (S. Diehl)

0167-739X/00/$ – see front matter
PII: S0167-739X(99)00093-X

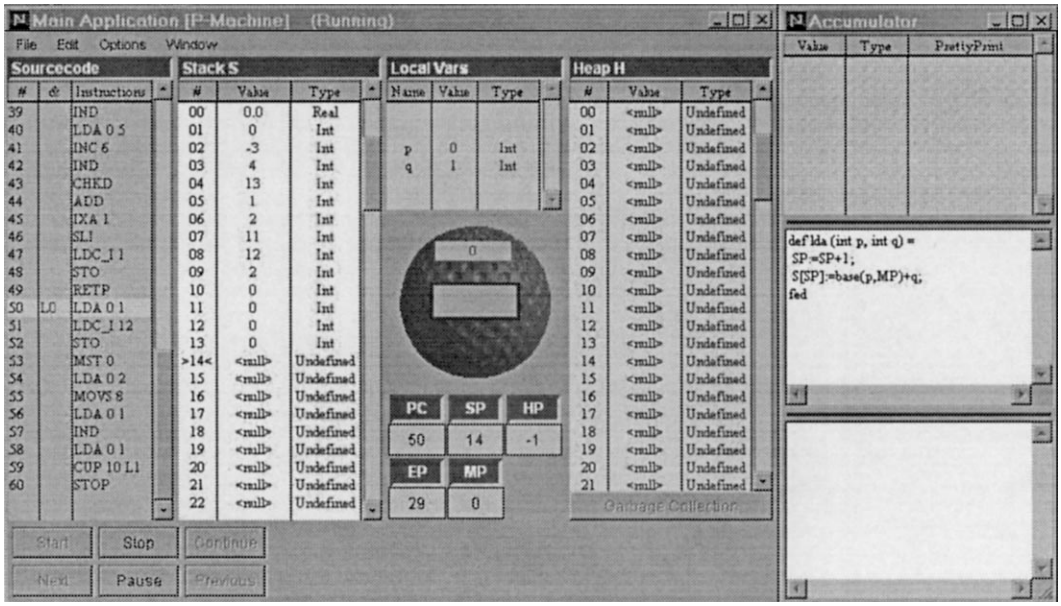

Fig. 1. Screenshot of an animated P-machine.

2. GANIMAM

GANIMAM is a web-based system which generates visualization of abstract machines from specifications, and animates the execution of abstract machine programs on these machines.

In the GANIMAM Java applet the user can select or edit a previously defined specification of an abstract machine or write a new one. Then this specification is sent to the server. A CGI script on the server generates Java code and using a Java compiler it translates this code into new class files. These files are then dynamically loaded into the running applet. In combination with the GANIMAM base package classes these class files form an interactive Java applet. The interaction of the system components is shown in Fig. 2. The user can enter machine programs, modify the layout of the different parts of the visualized abstract machines and control the animation of the execution of her abstract machine programs.

The automatic layout groups the different memories around the accumulator (the golfball in the middle). Source code and stacks are placed to the left, stacks to the right, local variables above and registers below the accumulator. Associated with the accumulator is an accumulator window, which shows the expressions which are currently evaluated and the definitions of the instructions or functions which are currently executed. Double clicking with the right mouse button at an instruction in the source code window, loads its definition into the accumulator window. Double clicking with the left mouse button at an instruction sets the value of the program counter to the address of that instruction, i.e. the execution of the abstract machine program continues at that address. Clicking at a cell of a stack, heap or register opens a window. In this window the user can change the value and type of that cell. For registers only the value can be changed.

3. Implementation

GANIMAM is a complex web application and combines many technologies. On the server, the compiler for the specification language was implemented with C, Lex and Yacc. Several CGI scripts had to be implemented, e.g. to access an existing compiler from Pascal to P-Code. Client/Server communication was implemented with the help of the Java Networking API. The AWT was used to implement the graphical user interface of the client and a new layout manager was implemented for the automatic and customizable layout of the abstract machine components. To save

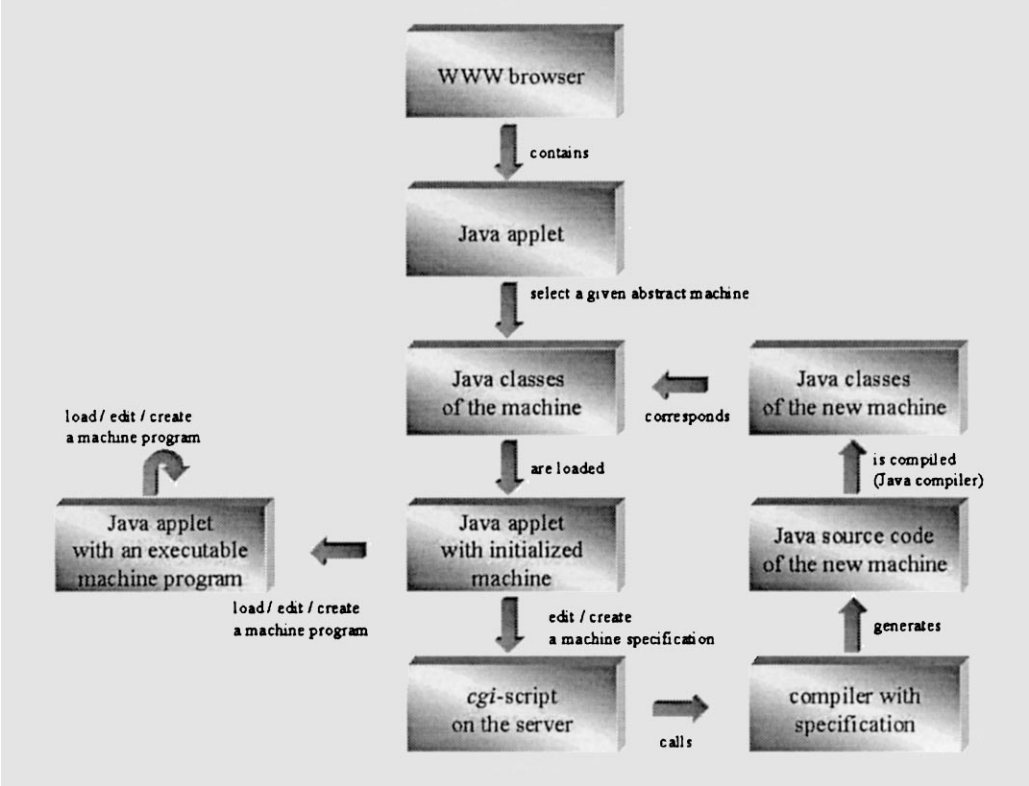

Fig. 2. Interaction of system components.

the customized layout locally on the client computer we use object serialization, certificates and privilege management. Garbage collection was implemented on the heaps of the generated machines. Last but not least, the Java Reflection API enabled us to load classes generated at runtime into the running applet.

4. Specification language

Finding low-level principles and casting them into language constructs is the first step towards a specification language. A well-designed specification language enables us to generate implemenations and visualizations. A crucial point of our specification language is that it applies to abstract machines for programming languages of different paradigms. Our specification language is based on the notation used in the compiler design text book by Wilhelm and Maurer [10] to define abstract machines for imperative, logical and functional programming languages. Recently the notation was also used to describe the Java Virtual Machine [5]. The core of our specification language is a control flow language with assignments, expressions, conditionals and loops. Control flow languages are a standard specification method for imperative, functional and logical programming languages, e.g. [10,1,8]. For the specification of abstract machines for functional languages sometimes rewriting rules have been used, e.g. for the CAM [7], but usually they can be easily reformulated in a control flow language [6].

At the heart of our specification language is a general machine model. An abstract machine consists of a set of instructions, a program store, heaps, stacks [1] and registers. The machine runs in a loop executing the instruction currently pointed at by a special register, the program counter (PC).

[1] In some abstract machines there are several stacks, e.g. in the WAM we have the environment stack, the trail and the PDL for recursive implementations of unification.

```
program_unit     → declarations definitions
declarations     → [declaration {; declaration}]
declaration      → REGISTER decl_REG | HEAP decl_HEAP
                   | STACK decl_STACK | OBJECT decl_OBJ
decl_REG         → IDENTIFIER [, decl_REG]
decl_HEAP        → array with IDENTIFIER {, array with IDENTIFIER }
decl_STACK       → array with IDENTIFIER
decl_OBJ         → IDENTIFIER ( components ) {, IDENTIFIER ( components ) }
components       → IDENTIFIER {, IDENTIFIER}
array            → IDENTIFIER [ CONSTANT ]
definitions      → {definition}
definition       → def command = instructions fed
                   | fun funcommand = instructions nuf
command          → IDENTIFIER arguments
funcommand       → type IDENTIFIER arguments
arguments        → [( [identifiers] )]
identifiers      → type IDENTIFIER {, type IDENTIFIER}
type             → int | boolean | address | pointer | real | ...
instructions     → [instruction {; instruction}]
instruction      → assignment | condition | for | case | call
                   | return expression | ie ( predefinedEvents )
predefinedEvents → markProcedureStackFrame ( register ,
                   CONSTANT ) | comment ( commenttext ) | ...
assignment       → lval := rval
lval             → register | memory
rval             → expression
condition        → if condit then instructions
                   [else instructions] fi
for              → for init downto expression do instructions od
                   | for init to expression do instructions od
init             → IDENTIFIER := expression
case             → case expression of expressionlists +esac
expressionlists  → expressionlist expressionlists | defaultlist
expressionlist   → CONSTANT : begin instructions end
defaultlist      → [otherwise : instructions]
call             → IDENTIFIER ( f_arguments )
f_arguments      → [expression {, expression}]
```

Fig. 3. Syntax of abstract machine specification language.

```
while(true){
  PC:=PC+1;
  execute instruction at CODE[PC-1]
}
```

In this model an abstract machine can be specified by declaring its heaps, stacks and registers and defining its instructions.

In Fig. 3 the syntax of our specification language is given. A specification starts with declarations of stacks, heaps and registers. Then auxiliary functions (with `fun`) and machine instructions (with `def`) are defined. Functions must be defined before they are used. The predefined datatypes currently include integers, booleans, reals, addresses and pointers. Addresses refer to positions in the program code, whereas pointers point to cells in the stacks or heaps. One could imagine to have pointers to registers, but we have not found an abstract machine which needs this. There is also a construct `OBJECT` to declare structured data types:

```
OBJECT CLOSURE (cp,gp),
VECTOR [ ] (v);
```

It was heavily used in the specification of the MAMA, a variant of the G-machine, which is used as a target architecture for functional programming

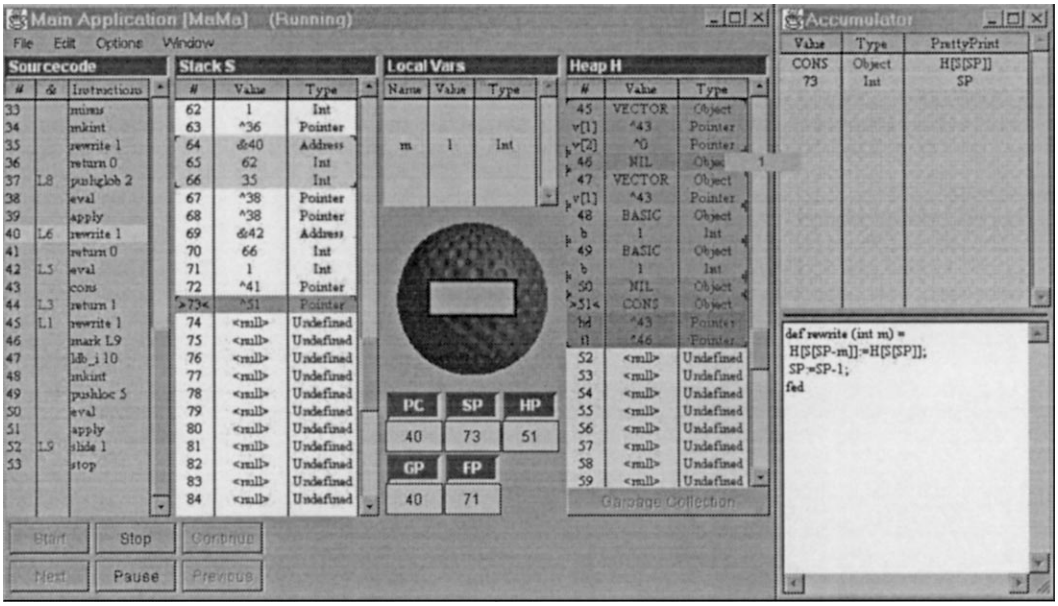

Fig. 4. Screenshot of the animated MAMA.

languages. In Fig. 4 instances of these structured data types are visualized in the heap window. The above example defines two structured data types: `CLOSURE` with the components `cp` and `gp` and `VECTOR` with a component `v` which is an array of values. With the function `new()` a new object is allocated and initialized on the heap and a tagged pointer to this object is returned. The tag is the name of the data type used.

```
def mkvec (int n) =
  S[SP-n+1] :=
    new (CLOSURE: S[SP], S[SP-1]);
  SP := SP-1;
fed

def eval =
  if H[S[SP]].tag =CLOSURE then
    S[SP+3] := S[SP];
    S[SP] := PC;
    S[SP+1] := FP;
    S[SP+2] := GP;
    SP := SP+3;
    FP := SP-1;
    GP := H[S[SP]].gp;
    PC := H[S[SP]].cp;
  fi
fed
```

The careful reader will notice, that labels are not part of the specification language, but that they occur in abstract machine programs (see source code window in Fig. 1). Labels are used instead of concrete addresses and a preprocessor contained in the runtime system of GANIMAM maps these labels onto concrete addresses.

As an alternative to initializing the state of an abstract machine by a sequence of abstract machine instructions, we allow more compact initialization sequences in the abstract machine code:

```
.init_prog
#
source: S
offsetregister: MP
0: 3.14 Real
1:−3 Int
2:true Boolean
#
```

In the above example the stack `S` is initialized as follows: `S[MP+0]` is set to 3.14 and its tag to `Real`, `S[MP+1]` is set to −3 and its tag to `Int` and `S[MP+2]` is set to `true` and its tag to `Boolean`. This means, that an abstract machine starts with all registers set to the default values given in the abstract

```
// declarations
STACK    S[100] with SP=-1;
HEAP    H[100] with HP=-1;
REGISTER  PC,MP,EP;

// specification of auxiliary functions
fun int base (int p, int dv) =  // compute static predecessor
 if (p=0) then                  // of the current procedure
  return dv;
 else
  return base(p-1,S[dv+1]);
 fi;
nuf

// specification of an instruction
def mst (int p) =     // create a procedure frame
 S[SP+2]:=base(p,MP); // pointer to frame of static predecessor
 S[SP+3]:=MP;         // pointer to frame of dynamic predecessor
 S[SP+4]:=EP;         // max. depth to evaluate expressions
 SP:=SP+5;            // procedure frame has at least 5 cells
fed
```

Fig. 5. Example specification

machine specification and stacks and heaps are empty and then the initialization sequence is executed to set the values of registers, stack and heap cells. But in addition such initialization sequences can occur throughout a machine program to put the machine into certain state. This feature is helpful for inspection and debugging of abstract machines.

4.1. An example specification

In Fig. 5 we show an excerpt of the specification of an abstract machine for imperative languages [10], a variant of the P-machine. In this example the instruction `mst` (mark stack) is defined which pushes a frame for a procedure on top of the stack. In the specification a stack `S`, a heap `H` and the register `PC`, `MP` and `EP` are declared. The stack and heap declarations also the special purpose registers `SP` and `HP` are declared, which point to the top of the currently used memory area. The special purpose register `PC` is automatically defined by GANIMAM and its declaration is optional. Next auxiliary functions are defined. Here it is a function, which computes the static predecessor of the current procedure, in the WAM for example such functions include `unify()` or `deref()`.

5. Visualizing principles

Similar principles are known by different names in different communities, e.g. stack frames are also known as activation records or choice points depending on the language paradigm, and programmed graph-reduction and programmed unification are two instances of the same, more general principle of implementing the execution of a binary operation by having one argument on the heap and the other argument encoded by the machine instructions.

One has to distinguish principles of the programming language, e.g. inheritance of methods in Java, and principles to implement these in an abstract machine, e.g. chains of method tables.

In the abstract machine we can only visualize the implementation principles. In addition, textual comments can explain the relation to the programming language principles. When visualizing a principle, we can visualize its different aspects:

- Show the information used and produced by the principle.
- Animate operations performed by the principle, e.g. dereferencing of variables in the WAM.
- Visualize properties and invariants enforced by the

```
def mst (int p) =
 ie(markProcedureStackFrame(SP,5));
 // Starting a cell S[SP], the following 5 cells are
 // graphically marked as a procedure frame
 S[SP+2]:=base(p,MP);
 S[SP+3]:=MP;
 S[SP+4]:=EP;
 SP:=SP+5;
fed
```

Fig. 6. Example of animation annotation in an instruction definition.

```
def mst (int p) =
 ie(comment("Initialize procedure stack frame at SP="+SP
            +" and PC="+PC));
 S[SP+2]:=base(p,MP);
 S[SP+3]:=MP;
 S[SP+4]:=EP;
 SP:=SP+5;
fed
```

Fig. 7. Example of a runtime comment in an instruction definition.

principle, e.g. in the WAM variables of higher addresses always reference that of lower addresses.

Some of the principles of an abstract machine are not explicit at the abstraction level of our specification language. For example stack frames are common to all abstract machines we considered. Stack frames are a means to implement recursion. Usually the stack cells of a stack frame are allocated by a sequence of one or more instructions, which push values on top of the stack.

Other instructions access information relative to the beginning of the stack frame or release the stack frame as a whole. [2] There is no single construct in our specification language to allocate a stack frame. When visualizing an abstract machine it is important that we do not only draw low-level abstractions captured by our specification language constructs, but also higher-level abstractions. In order to do this we added visualization annotations to our specification language. These are implemented as calls to a method `ie()`, see example in Fig. 6. These annotations can be compared to interesting events in some algorithm animation systems [3].

A very general and useful annotation is a runtime comment, see example in Fig. 7. It produces a textual output which is shown in a console window. Using runtime comments the output in the console window can be used as a trace of the execution of the abstract machine, see Fig. 8.

6. The benefits of interactive animations

GANIMAM provides several ways of user interaction. First the user can enter or modify the specification of an abstract machine. After generating an

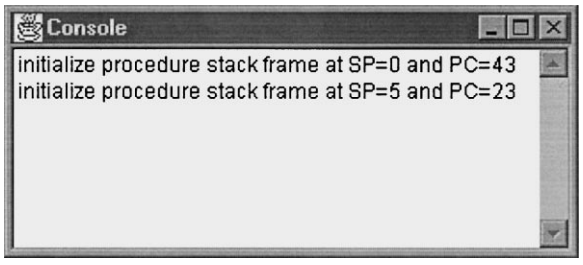

Fig. 8. Screenshot of console window.

[2] In the WAM stack frames are called environments and a special optimization called environment trimming decreases the number of stack cells of an environment during its live span.

implemenation of the abstract machine, the user can input an abstract machine program, execute it step by step and inspect the contents of each register or memory cell. When executing an instruction animations show the flow of information from registers or memory cells to a conceptual operation unit, called accumulator, and from the accumulator back to registers or memory cells. The evaluation done in the accumulator is shown in a special window, see Fig. 9.

Annotations only help to visualize principles which we know upfront. GANIMAM can also be used to detect new principles by experimenting with specifications and abstract machine programs. Such an experimental approach can be used for two purposes:

- As part of an explorative learning software it enables students to formulate hypotheses and validate or invalidate them by changing specifications or abstract machine programs. Additional text guides the learner, to make sure she does not miss the important issues. Such issues could be caller-save-registers vs. callee-save-registers, finding the frame of the static predecessor or lazy vs. eager evaluation.
- As a development tool it can help to detect errors and optimizations. As an example of such an optimization consider tail recursion optimization. By tracing the execution of example programs it might become apparent that the information stored in a frame is not needed after certain recursive calls.

GANIMAM is not meant to replace classical teaching or development approaches, but to supplement and enhance these. GANIMAM can also be used by researchers to present their new implementation techniques or for rapid prototyping.

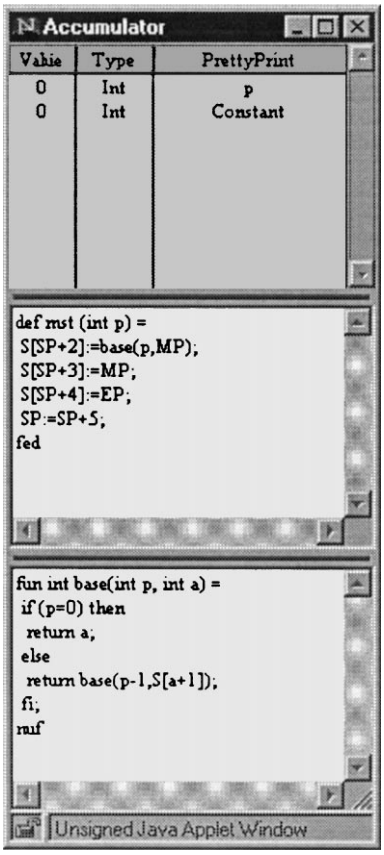

Fig. 9. Screenshot of the accumulator window.

7. Current and future work

In the GANIMAL project[3] we will also develop generators for interactive animations of other compiler phases. We are currently looking into how to evaluate the software produced in the GANIMAL project. Those evaluations of algorithm animations we are aware of [4,9] lack a serious approach both for collecting and evaluating the data. To avoid these problems we plan to cooperate with cognitive psychologists.

In the current version of GANIMAM structured datatypes like records, objects or ML datatypes can be built with the help of pointers but a coherent visualization does not yet exist. In a later version we will use the graph layouter which is currently under development in the GANIMAL project to visualize structured data types. It should also be very easy to extend GANIMAM to visualize concurrent (one machine scheduling different tasks) and parallel execution (several machines running at the same time) and their communication behavior.

8. Conclusion

We introduced GANIMAM, a web-based system to generate interactive animations of abstract machines

[3] This project is funded by the Deutsche Forschungsgemeinschaft and started in summer 1998.

from specifications. During the development of GANIMAM common principles of abstract machines have been considered at three levels: the design of the specification language, the choice of graphical annotations to visualize higher-level abstractions and the use of the system to explore and better understand known and detect new principles. Our final goal is to integrate GANIMAM into a learning software for compiler design and thus enabling students to solve exercises related to abstract machines by an experimental and explorative approach. GANIMAM can be accessed from the web page of the GANIMAL Project at the University of Saarland: http://www.cs.uni-sb.de/GANIMAL/.

Acknowledgements

The authors thank Andreas Placzek, who helped to implement a first prototype of GANIMAM.

References

[1] H. Aït-Kaci, Warren's Abstract Machine — A Tutorial Reconstruction, MIT Press, Cambridge, MA, 1991.
[2] A. Martin, F. Martin, Generation of Efficient Interprocedural Analyzers with PAG, in: A. Mycroft (Ed.), SAS'95, Static Analysis Symposium, Lecture Notes in Computer Science, vol. 983, Springer, Berlin, 1995, pp. 33–50.
[3] M.H. Brown, Algorithm Animation, MIT Press, Cambridge, MA, 1987.
[4] M.D. Byrne, R. Catrambone, J.T. Stasko, Do algorithm animations aid learning, Technical Report GIT-GVU-96-18, Georgia Institute of Technology, 1996.
[5] S. Diehl, A Formal Introduction to the Compilation of Java, ractice and Experience 28 (3) (1998) 297–327.
[6] J. Hannan, Making abstract machines less abstract, Proceedings of FPCA'91, Lecture Notes in Computer Science, vol. 523, Springer, Berlin, 1991, pp. 618–635.
[7] M. Mauny, A. Suarez, Implementing functional languages in the categorial abstract machine, International Conference on LISP and Functional Programming, 1986.
[8] St. Pemberton, M. Daniels, Pascal Implementation, The P4 Compiler, Ellis Horwood, Chichester, UK, 1982.
[9] J.T. Stasko, Using student-built algorithm animations as learning aids, Technical Report GIT-GVU-96-19, Georgia Institute of Technology, 1996.
[10] R. Wilhelm, D. Maurer, Compiler Design: Theory, Construction, Generation, Addison-Wesley, Reading, MA, 1995.

Thomas Kunze is a student of computer science and economics at Saarland University, Germany. He is member of the GANIMAL team at the research group of Prof. Reinhard Wilhelm and has implemented the abstract machine visualization as part of his master thesis. Thomas Kunze is interested in Java programming and software engineering in general.

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 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, visualisation, Java and VRML. He teaches courses and seminars at university and in industry about these topics. Since summer 1998 he is project leader of the project GANIMAL sponsored by DFG (German Research Council).