An Abstract Machine for an Object-Oriented Language with Top-Level Classes

Technischer Bericht Nr. A 02/94

Christoph Böschen
Christian Fecht
Andreas V. Hense
Reinhard Wilhelm

Fachbereich 14
Universität des Saarlandes

Technischer Bericht Nr. A 02/94

An Abstract Machine for an Object-Oriented Language with Top-Level Classes

Christoph Boeschen
Christian Fecht
Andreas V. Hense
Reinhard Wilhelm

Universität des Saarlandes
Postfach 151150
66041 Saarbrücken
{boeschen,fecht,hense,wilhelm}@cs.uni-sb.de

March 11, 1994

Abstract

Object-oriented programming languages where classes are top-level, i.e. not first-class citizens, are better suited for compilation than completely dynamic languages like SMALLTALK or SELF . In O'SMALL, a language with top-level classes, the compiler can statically determine the inheritance hierarchy. Due to late binding, the class of the receiver of a message must be determined at run time. After that a direct jump to the corresponding method is possible. Method lookup can thus be done in constant time.

We present an abstract machine for O'SMALL based on these principles. It is a concise description of a portable O'SMALL implementation.

1 Introduction

Abstract machines have been used for essentially two purposes, to ease the language implementation task and to make an implementation of a programming language portable.

Real machines, those available on the market, closely reflect the features of imperative languages. Abstract machines bridge the gap between high-level programming languages and the machine code of existing real machines. The instruction set of an abstract machine is chosen such that

- each instruction can be implemented by a handful of instructions on a real machine

```
class Account inheritsFrom Base
def var b := 0
in
   meth balance()    b
   meth credit(n)    self.transact(n)
   meth debit(n)     self.transact(-n)
   meth transact(x) b := b + x
ni

class PAccount inheritsFrom Account
def var fee := 5
in
   meth transact(x) super.transact(x - fee)
ni

class EAccount inheritsFrom PAccount
meth balance()  self.transact(0);
                super.balance
```

Figure 1: Class definitions

- and the high-level language constructs can be translated in a simple and concise way.

Abstract machines are a means to make implementations portable. A package consisting of a Pascal compiler written in Pascal and compiling to the P–Machine [14], together with an assembler and an interpreter for P–code, both also written in Pascal, was the basis for the worldwide distribution of the Zürich Pascal implementation. All that was required to port this implementation to a new architecture, was the emulation or the compilation of the P–Machine.

The construction of an abstract machine is a design process without any theorems or proofs. There have been attempts to formally deduce abstract machines from given semantics of various languages [1, 10, 12, 13] but they neither developed new machines nor could the deductions be proved correct automatically. We believe that, if an abstract machine is well designed and easy to understand, the faith in its correctness is easier to gain than the faith in the correctness of a long series of transformations. After all, there can also be bugs in proofs.

SMALLTALK was equipped with an abstract machine, namely the byte code machine of Goldberg et al. [5]. Since then, the compilation schemes of new SMALLTALK versions have considerably deviated from the original machine. However, we are not aware of any more recent publications on abstract machines for object-oriented programming languages. There have been papers on compilers. Johnson et al. [9] present an optimizing compiler for SMALLTALK which makes use of type declarations that have been added to the language. Chambers et al. [2, 3, 4] worked on the implementation of SELF, a language that is even more dynamic than SMALLTALK.

We present an abstract machine in the tradition of the MaMa [15, 16] and the P-Machine. The source language for the compiler is O'SMALL [6], a dynamically typed object-oriented programming language where classes have a more static nature than in SMALLTALK. Using the same framework as for imperative and functional languages it is easy to see the differences and the similarities between the three language groups.

The paper is structured as follows. Section 2 introduces O'SMALL by an example program. Section 3 gives an overview of the parts of the abstract machine without going into details. Section 4 is the detailed description of the abstract machine in form of the translation function. Here, the standard issues are presented before the more interesting object-oriented issues to give the reader a chance to get used to the functioning of the abstract machine. The reader familiar with object-oriented programming may skip section 2. The reader familiar with abstract machines may skip sections 4.1 through 4.3.

2 The Language O'SMALL

O'SMALL is a simple object-oriented language that can best be compared to SMALLTALK. Like SMALLTALK it is class-based and uses pseudo-variables `self` and `super` for inheritance. Classes in O'SMALL are not first-class citizens like in SMALLTALK, they are top-level. We will discuss this important difference in the remainder of this work. The syntax is different and the concept of objects is not as strictly advocated as in SMALLTALK: e.g. there are primitive data types like boolean, integer and the like.

We will explain the semantics of O'SMALL by an example program (Fig. 1) and use an informal operational method-lookup semantics. A formal semantics can be found in [6]. The example program in Fig. 1 contains three class declarations. The resulting inheritance hierarchy is contained in Fig. 3. O'SMALL uses simple inheritance. Multiple inheritance can be expressed by explicit wrappers [7]. These features will be discussed later. Class `Account` inherits from the base class. The latter is essentially the empty class where no methods are declared. Variable `b` is the instance variable of objects of class `Account`. Instance variables contain the internal state of objects. They can only be accessed by the methods of the class that declares them, *not* by methods of subclasses. Thus we have *encapsulated instance variables.* Instance variables are always initialized: There are no "nil" variables. The methods of the class together with the methods of the ancestor classes make up the interface of the object. In this case we have four methods because the ancestor class is empty. Method `balance` returns the value of the instance variable `b`. Method `credit` sends the message `transact` to `self`, i.e. the object itself.

Fig. 2 shows an object `a` that belongs to class `Account`. If a message is sent to `a`, we start looking for a method with a corresponding name in class `Account`. If no method with that name is found, we continue our search in the superclass. This goes on until we reach the root of the inheritance tree, i.e. the base class. Since the base class is empty we know upon our arrival there that no corresponding method has been found. This means an error. This process is called *method lookup* and we speak of *method lookup semantics* when we use this method for explaining an

```
def var a := new Account
    var b := new Account
    var p := new PAccount
    var e := new EAccount
in
   a.credit(5000);
   output(a.balance);                 {   ---->  5000   }
   p.credit(5000);
   output(p.balance);                 {   ---->  4995   }
   e.credit(5000);
   output(e.balance);                 {   ---->  4990   }
ni
```

Figure 2: The main program

object-oriented language. We will come back to method lookup when we discuss the main program in Fig. 2. Method **transact** changes the internal state of the object. O'SMALL is a language for studying the essence of the inheritance mechanism. In order to keep the language concise and thus manageable there is no elaborated visibility concept for methods. In this example one would like the visibility of the method **transact** to be constrained to this class and its descendants. Access from outside should be limited to the first three methods.

The bank that used these accounts was in the red and one day a clever consultant proposed to charge a fee for every operation on an account. Since the system was programmed in an object-oriented way the change was quite simple. All that had to be done was the creation of a subclass **PAccount** and create all new accounts as members of this new class. Objects of this new class have an additional instance variable **fee**. They inherit the instance variable **b** but, as already said, **b** is invisible for the new methods. The transaction method is overwritten. With **super** we are able to retrieve the method of the nearest ancestor class and thus the method that has just been overwritten. Usually messages are sent to objects. Messages to **super** are an exception. They are sent to the object itself but the method lookup starts in the class above the class where the method that uses **super** is declared. Therefore, the corresponding method to a message sent to **super** can be statically determined while the method of a message sent to **self** cannot.

The main program that follows the class declarations is contained in Fig. 2. Four objects are created. The message **credit(5000)** is sent to **a**. This results in the message **transact(5000)** to **a**. The internal state of **a** is set to 5000 and this value is output in the next line. The message **credit(5000)** is also sent to **p**. This results in the message **transact(5000)** to **p**. This time the method **transact** of class **PAccount** is found. Therefore, the internal state of **p** is set to 4995 and it is this value that is output in the next line.[1]

Before all the clients ran away, the bank introduced expensive accounts in accordance with the theory that there is no observation without destruction. The reader

[1] In O'SMALL, statements and statement lists (Fig. 1) always return a value. Unlike SMALLTALK, which has an explicit return statement, O'SMALL methods return the "last value".

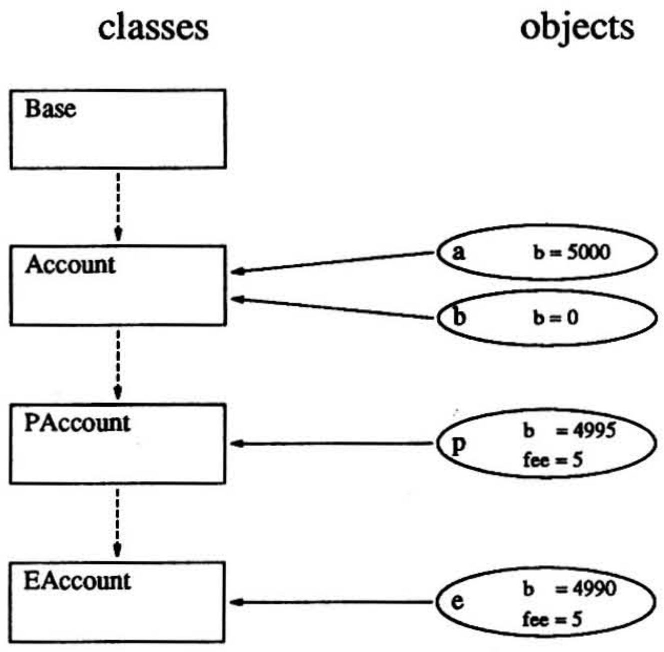

Figure 3: Inheritance hierarchy

may check that the output for e results in 4990.

The abstract syntax of O'SMALL is given in Fig. 4. Brackets stand for list with at least one element. Statement lists are written with semicolons like in ordinary imperative languages. If a class defines no new instance variables we omit **def** and ni (cf. Fig 1 class EAccount).

Compilers try to compute a large amount of information in advance in order to reduce the overhead at run time. We will discuss here which features can be computed in advance (*static features*) and which have to be computed at run time (*dynamic features*) in O'SMALL. As opposed to SMALLTALK, where classes are first-class citizens, O'SMALL classes are static. We believe that classes should be of a longer duration than objects. Objects are created, change their states, and die eventually by garbage collection. Classes describe the world and should change less frequently. They should have the same characteristics as modules in other languages (e.g. SML [11]). If a message is sent to an object we do not know which class the object belongs to. Therefore, we have to look up the class of the object at run time (Fig. 3). However, once we have the class of the object we can immediately find the corresponding method. The "search" for the method through the ancestors can be done at compile time because classes cannot change at run time. This is different from SMALLTALK, where in principle all ancestors have to be searched because one does not know whether the class hierarchy has changed or not.

$$
\begin{array}{llll}
\text{p} & ::= & [class]\ [s] & \\
\text{class} & ::= & \texttt{class}\ c_1\ \text{inheritsFrom}\ c_2 & \\
 & & \texttt{def var}\ v_1\ :=\ e_1 & \\
 & & \qquad\qquad \vdots & \\
 & & \qquad \texttt{var}\ v_n\ :=\ e_n & \\
 & & \texttt{in}\ \ meth_1 & \\
 & & \qquad \vdots & \\
 & & \qquad meth_k & \\
 & & \texttt{ni} & \\
\text{meth} & ::= & \texttt{meth}\ m(x_1,\ldots,x_n)\ [s] & \\
\text{s} & ::= & e & \\
 & | & \texttt{if}\ e\ \texttt{then}\ [s]\ \texttt{else}\ [s]\ \texttt{fi} & \\
 & | & v\ :=\ e & \\
 & | & \texttt{output}\ e & \\
 & | & \texttt{while}\ e\ \texttt{do}\ [s]\ \texttt{od} & \\
 & | & \texttt{def var}\ v_1\ :=\ e_1,\ldots,\ \texttt{var}\ v_n\ :=\ e_n\ \texttt{in}\ [s]\ \texttt{ni} & \\
\text{e} & ::= & i & \text{(integer)} \\
 & | & b & \text{(booleans)} \\
 & | & \texttt{self} & \\
 & | & v & \text{(variables)} \\
 & | & op_{un}\ e & \\
 & | & e_1\ op_{bin}\ e_2 & \\
 & | & \texttt{new}\ c & \\
 & | & e.m(e_1,\ldots,e_n) & \\
 & | & \texttt{super}.m(e_1,\ldots,e_n) &
\end{array}
$$

Figure 4: Abstract Syntax of O'SMALL

3 The Abstract Machine

In this section we give a brief overview of the abstract machine. We will describe which memory areas are used by the machine, how they are structured and which registers point to them. The exact functioning will become clearer in later sections. The abstract machine consists of the following three memory areas:

- the program store PS, containing the translated program as a sequence of machine instructions. The program counter PC points into the program store.
- the stack ST, where evaluation takes place, with stack pointer SP and frame pointer FP.
- the heap HE, where representations of objects are stored. The Heap pointer HP points to the last occupied cell of the heap.

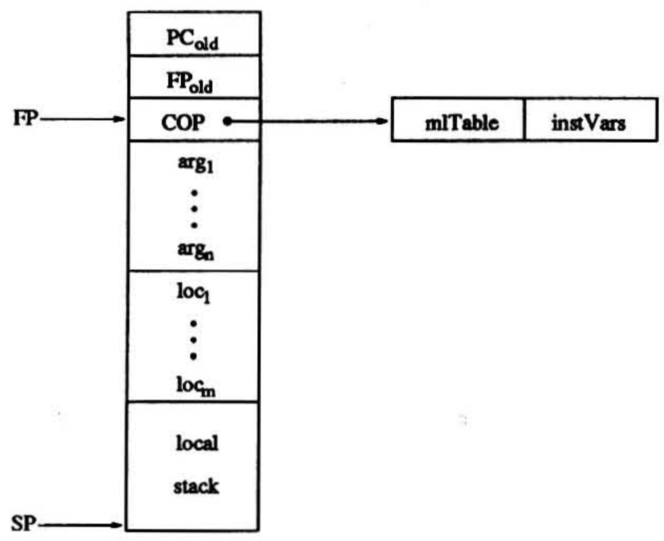

Figure 5: Stack frame

3.1 The Program Store

The program store is an array of instructions. Each instruction consists of an opcode and an optional list of operands. Initially, the program counter PC is set on the first instruction in the program store. In order to execute the program, the following cycle is continually executed:

- load the current instruction (this is the instruction pointed to by the PC).
- increment the PC by one.
- interpret the loaded instruction.

The machine halts if during execution the instruction **halt** is encountered.

3.2 The Stack

The evaluation stack ST is an array of stack cells. A stack cell can contain one of the following objects:

- an integer or boolean
- a reference to the heap
- a reference to the program store
- a reference to the stack

Therefore, stack cells must have a tag conveying the intended meaning of their content. For the sake of simplicity, we will omit these tags in the description of the abstract machine.

The topmost stack cell is pointed to by the stack pointer SP. The stack itself is divided into so-called *stack frames*, each frame corresponding to a method call, or, stated otherwise, corresponding to an object to which a message was sent. The frame pointer FP gives access to the topmost frame. A frame has the following structure (see Fig. 5):

- Organizational cells

  1. The continuation address $\mathrm{PC}_{\mathrm{old}}$. This is the address in the program store, where execution has to be continued after returning from the method corresponding to the frame.
  2. $\mathrm{FP}_{\mathrm{old}}$, the saved FP, is a pointer to the previous frame.
  3. The current object pointer (COP): It points into the heap to that object to which the current message was sent.

  These cells are installed by the caller of the method, i.e. by the sender of the message. The first two of them serve to restore the right context after completion of the method call. The third component (COP) gives access to the current object's method lookup table and its instance variables.

- A local environment consisting of the arguments of the message and of local variables introduced by `def`-statements. Note that all elements of the local environment can be addressed via FP.

- A local stack where expressions are evaluated.

3.3 The Heap

Representations of objects are stored in the heap. Each heap cell contains an entire object, i.e. a method lookup table and a vector of instance variables. The method lookup table is only conceptually contained in each object. In a real implementation, there would simply be a reference to the method lookup table of the class. Since an object can have an arbitrary number of instance variables a heap cell can be arbitrarily big.

There are two things that can happen to an object. Messages can be sent to objects and the methods of the object's class can access its instance variables. The set of all instance variables is known for each class at compile time. Therefore, one component of the representation of an object must be a vector containing the values of the instance variables of the object.

Assume that a message m with some arguments is sent to an object. Of course, this message m should denote a method with name m. The body of this method has to be evaluated in the context determined by the arguments of the message m and the object to which the message was sent. In imperative languages with static binding, such as PASCAL, if a procedure p is called, we can determine at compile time which procedure is actually meant by p, i.e which procedure is *bound* to p. In object-oriented languages another mechanism for binding, called *late-binding*, is used. Here, the object to which message m was sent itself decides which method is meant by m. This is usually called method lookup. Since message sending is crucial in object-oriented programming, the dynamic assignment of actual methods to messages should be as efficient as possible. In O'SMALL, however, class structure is static and cannot change at run time. Thus, for every class c we can determine at compile time which message m is understood by objects from c and which method is assigned to m. With every class c, one can associate a function

$$f_c : M \rightarrow \mathit{Label}$$

where *Label* denotes the set of indices into the program store. For a message m, $f_c(m)$ denotes the beginning of the translation of the method that is assigned to m. How f_c is computed for every class c will be described in more detail in a later section when describing how class definitions are translated. From this follows that the second component of the representation of an object must be a representation of the function f_c, provided c is the class of the object. Functions f_c are represented by the abstract data type *methodLookupTable*. We assume that the following two functions are available on *methodLookupTable*:

$$\begin{array}{lcl} \mathit{makeTable} & : & 2^{\mathbf{Meth}} \rightarrow \mathit{methodLookupTable} \\ \mathit{lookup} & : & M \rightarrow \mathit{LABEL} \end{array}$$

An object is therefore represented in the heap as a structure

$$(\mathit{mltable} : \mathit{methodLookupTable}, \mathit{instVars} : \mathit{array}[1..n])$$

3.4 Method Lookup

Looking at Fig. 4 we see that a class definition consists of a new class name, the name of the superclass, a vector of instance variables together with their initialization, and a list of methods. We can regard the whole set of a program's class definitions as a tree (inheritance hierarchy) with the base class as its root. What kind of information do we need about classes at run time?

Since we have top-level classes in O'SMALL, we know that the inheritance hierarchy does not change at run time. Regardless of the choice of semantics, be it wrapper semantics [6] or method lookup semantics [5], we can calculate the method lookup tables in advance. There is one table per class.[2] A method lookup table is a partial mapping from message selectors to methods. We forget the inheritance tree and unfold the implicit contents of the class.

In addition to the methods we need the set of instance variables and the expressions that are used to initialize them. As already mentioned, we have encapsulated instance variables, i.e. their name spaces are disjoint and they can only be accessed in the class where they are declared and *not* in any subclass. This is different from SMALLTALK.

This concludes the short overview of the machine. The next section reveals all the details. While reading this section it is useful to permanently have access to Fig. 5.

[2]In this description, each object has its own table. Since the tables of objects of the same class are identical, one shares them in the implementation.

4 The Translation Function

For every syntactic category, there is a corresponding translation function that translates elements of this category into code, i.e. sequences of instructions. Expressions, for instance, are translated by the function *codeE* into code which, when executed, computes the value denoted by the expression. The instruction set of our abstract machine can be found in Fig. 6.

In order to generate instructions that access variables correctly — either by reading or writing them —, almost every translation function has an address environment as additional parameter. The set of variables visible at every point can be divided into local variables - these are formal arguments of methods or variables introduced by `def`-statements - and instance variables of objects. Furthermore, these two sets are known at compile time, so that addressing can be accomplished by arbitrarily arranging the variables in these sets. At run time, local variables are stored in a continuous block of the current frame, whereas instance variables are stored on the heap in the vector of the current object. Local variables are addressed via the FP register, instance variables are addressed relative to the beginning of the vector. An address environment β is thus a function

$$\beta : V \rightarrow \{LOC, INST\} \times \mathbb{N}$$

mapping a variable v to a pair (k, a). The first component $k \in \{LOC, INST\}$ tells us, whether v is a local or an instance variable, i.e. whether v is stored in the current frame or in the current object. The second component a gives the position relative to the beginning of the respective memory area.

We write address environments β as lists of bindings $[v_i \rightarrow (k_i, a_i)]_{i=1}^{n}$. We denote the set of all address environments by *AdrEnv*. The notation $\beta[v_i \rightarrow (k_i, a_i)]_{i=1}^{n}$ denotes an environment where the bindings to the variables v_i, $i = 1, \ldots, n$ in β get overwritten by the newer bindings $v_i \rightarrow (k_i, a_i)$.

It is now time to give an overview of all the translation functions we will use below in describing the translation process. The result of every translation function is a code sequence, i.e. an element of *Code*.

- $codeP : P \rightarrow Code$
  translates whole programs into code sequences.

- $codeC : Class \rightarrow Code$
  translates class definitions into code sequences.

- $codeM : Meth \times AdrEnv \rightarrow Code$
  translates a method *meth*. In the body of *meth* you can refer to the instance variables that were introduced in the same class definition as *meth*. Therefore, *codeM* needs an address environment as additional argument to deal with these variables.

- $codeS : S \times AdrEnv \times Bool \times Int \rightarrow Code$
  translates statements in a given address environment. Note, that in O'SMALL statements also denote values. The third argument of *codeS* indicates whether this value is needed after the execution of the statement. The fourth argument

instruction	definition	comment
pushunit	$SP := SP + 1;$ $ST[SP] := unit;$	push the value *unit* onto the stack
pushbool b	$SP := SP + 1;$ $ST[SP] := b;$	push a boolean onto the stack
pushint i	$SP := SP + 1;$ $ST[SP] := i;$	push an integer onto the stack
pushloc i	$SP := SP + 1;$ $ST[SP] := ST[FP + i];$	push a local variable onto the stack
storeloc i	$ST[FP + i] := ST[SP];$	store a local variable
pushinst i	$SP := SP + 1;$ $ST[SP] := HE[ST[FP]].instVars[i];$	push an instance variable onto the stack
storeinst i	$HE[ST[FP]].instVars[i] := ST[SP];$	store an instance variable
output	$print(ST[SP]);$ $ST[SP] := unit;$	print the top element of the stack
ujmp l	$PC := l;$	go to label l
jfalse l	*if* $ST[SP] = false$ *then* $PC := l$ *fi*; $SP := SP - 1;$	jump on false
pop n	$SP := SP - n;$	pop n elements from the stack
slide n	$ST[SP - n] := ST[SP];$ $SP := SP - n;$	slide up a value on the stack
mark	$ST[SP + 2] := FP;$ $SP := SP + 2;$	create part of new stack frame
call l_m, n	$FP := SP - n;$ $ST[FP - 2] := PC;$ $PC := l_m;$	go to the code of a method

send m, n	$FP := SP - n;$
	$ST[FP - 2] := PC;$
	if $newAdr = undef$
	  then error "Method not found"
	  else $PC := newAdr$ *fi*;
	where $newAdr = lookup(HE[ST[FP]].mltable, m)$
		send a message m with n arguments

return	$PC := ST[FP - 2];$ $ST[FP - 2] := ST[SP];$ $SP := FP - 2;$ $FP := ST[FP - 1];$	give up a stack frame
makeobject *table,n*	$HP := HP + 1;$ $HE[HP] := (\ mltable = table,$ $instVars = < \underbrace{undef, \ldots, undef}_{\text{ntimes}} > \ );$ $SP := SP + 1;$ $ST[SP] := HP;$	create a new object

Figure 6: Machine instructions

contains the size of the local environment. The size is needed for creating correct addresses during the translation of **def**.

- *codeSL* : $S^+ \times AdrEnv \times Bool \times Int \rightarrow Code$
  translates sequences of statements.
- *codeE*: $E \times AdrEnv \rightarrow Code$
  translates expressions in a given address environment. As mentioned above, an expression e is translated into a code sequence, whose execution leaves the value of e on the stack.

4.1 Simple Expressions

Integers and booleans are translated into instructions **pushint** and **pushbool** that load integers or booleans onto the stack, respectively. For every unary and binary operator there is a corresponding machine instruction, which is not included in Fig. 6. The operands of an operator expression are evaluated first. The machine instruction related to the operator takes them from the stack and replaces them by the result of the operation.

$$\begin{array}{lcl}
codeE\ i\ \beta & = & \mathbf{pushint}\ i \\
codeE\ b\ \beta & = & \mathbf{pushbool}\ b \\
codeE\ (op_{un}\ e)\beta & = & codeE\ e\ \beta; \\
 & & \mathbf{op}_{un} \\
codeE\ (e_1\, op_{bin}\ e_2)\ \beta & = & codeE\ e_1\ \beta; \\
 & & codeE\ e_2\ \beta; \\
 & & \mathbf{op}_{bin}
\end{array}$$

4.2 Variables and Assignment

A variable v occurring in an expression has to be translated into a machine instruction that loads the value assigned to v in the current run time environment onto the stack. The address environment β tells us where to find the value of v.

$$codeE\ v\ \beta \quad = \quad getVar\ v\ \beta$$

The auxiliary function *getVar* looks up v in the address environment and generates the appropriate instruction. The instruction **pushloc** loads a variable from the local environment in the current frame onto the stack, whereas **pushinst** loads instance variables from the current object onto the stack.

$$getVar\ v\ \beta = \begin{cases} \mathbf{pushloc}\ i & if\ \beta(v) = (LOC, i) \\ \mathbf{pushinst}\ i & if\ \beta(v) = (INST, i) \end{cases}$$

An assignment $v := e$ as a statement is compiled by the function *codeS*. We proceed as follows: we first translate the expression on the right-hand side of the assignment, thus computing the value of e. After that, the value of e lies on top of the stack and has to be moved to the location designated to v in the address environment.

If the value of the whole statement is not needed — this is indicated by the third argument being false, we can safely pop the value of e off the stack, otherwise we have to push 'unit' onto the stack.

```
codeS (v := e) β valueNeeded locVars =
        codeE e β;
        storeVar v β;
        pop 1;
        unit valueNeeded
```

Since in O'SMALL every statement leaves a value on the stack, we must also leave something after an assignment. According to the semantics of O'SMALL we leave a dummy value called 'unit' on the stack. This is expressed by the following auxiliary function.

$$\mathit{unit}\ \mathit{valueNeeded} = \begin{cases} \textbf{pushunit} & \mathit{if}\ \mathit{valueNeeded} \\ \epsilon & \mathit{otherwise} \end{cases}$$

As above, an auxiliary function *storeVar* looks up v in β and generates the appropriate instructions.

$$\mathit{storeVar}\ v\ \beta = \begin{cases} \textbf{storeloc}\ i & \mathit{if}\ \beta(v) = (\mathit{LOC}, i) \\ \textbf{storeinst}\ i & \mathit{if}\ \beta(v) = (\mathit{INST}, i) \end{cases}$$

4.3 Complex Statements

We present the translation of conditionals, while-loop, output statements and expression as statements without much ado.

```
codeS (if e then sl₁ else sl₂ fi) β valueNeeded locVars =
        codeE e β;
        jfalse l₁;
        codeSL sl₁ β valueNeeded locVars;
        ujmp l₂;
    l₁ : codeSL sl₂ β valueNeeded locVars;
    l₂ :
```

Conditionals and loops are translated by the usual structure of labels familiar to all writers of assembler programs.

```
codeS (while e do sl od) β valueNeededlocVars =
    l₁ : codeE e β;
        jfalse l₂;
        codeSL sl β false locVars;
        ujmp l₁;
    l₂ : unit valueNeeded
```

Also when a value is output, we leave a dummy value on the stack if the value is needed. Otherwise, the dummy value is directly popped off the stack again.

```
codeS (output e) β true locVars =
      codeE e β;
      output

codeS (output e) β false locVars =
      codeE e β;
      output;
      pop 1
```

A statement can also be an expression. There are two ways of translating it depending on whether the value is needed or not.

```
codeS e β true locVars =
      codeE e β

codeS e β false locVars =
      codeE e β;
      pop 1
```

Now, we come to the less familiar constructs. A **def**-statement extends the current environment by adding bindings to the variables v_i, $i = 1, \ldots, m$.

```
codeS ( def var v_1 := e_1
                 ⋮
            var v_m := e_m
        in
            sl
        ni ) β valueNeeded locVars =
        codeE e_1 β;
        codeE e_2 β[v_1 → (LOC, locVars + 1)];
                   ⋮
        codeE e_m β[v_i → (LOC, locVars + i)]_{i=1}^{m-1};
        codeSL sl β[v_i → (LOC, n + i)]_{i=1}^{m} (locVars + m);
        removeLocVars m valueNeeded
```

The variables v_i are initialized with the values of the expressions e_i. The expressions e_i are evaluated one by one leaving their values on the stack. The fourth parameter *locVars* of *codeS* contains the size of the local environment before the execution of the **def**-statement. After the evaluation of e_i the value of e_i lies at position $FP + locVars + i$. Thus, v_i is assigned the address $(locVars, n + i)$. Note that expression e_i can refer to variables v_j with $j < i$. Therefore, we must gradually adjust the address environment when translating the e_i's. Finally, the statement list *sl* is executed in the environment containing all local variables. After the execution of *sl* the environment for the v_i's has to be removed from the stack. If the value of the **def**-statement is not needed, this can be achieved by simply decrementing the stack pointer SP by m. Otherwise, the execution of *sl* has computed a value on the stack. The instruction **slide** m moves this value m cells up the stack.

$$\mathit{removeLocVars}\ n\ \mathit{valueNeeded} = \begin{cases} \textbf{slide}\ n & \textit{if valueNeeded} \\ \textbf{pop}\ n & \textit{otherwise} \end{cases}$$

In a statement list $s_1, \ldots, s_n$, the values of $s_1, \ldots, s_{n-1}$ are definitely not needed. The value of s_n is needed if the value of the whole sequence is needed.

$$\begin{aligned}
&\textit{codeSL}\ (s_1, \ldots, s_n)\ \beta\ \textit{valueNeeded locVars} = \\
&\qquad \textit{codeS}\ s_1\ \beta\ \textit{false locVars}; \\
&\qquad\qquad \vdots \\
&\qquad \textit{codeS}\ s_{n-1}\ \beta\ \textit{false locVars}; \\
&\qquad \textit{codeS}\ s_n\ \beta\ \textit{valueNeeded locVars}
\end{aligned}$$

For the main program, the classes are translated consecutively. The function *codeSL* is called with an empty local environment. The final value of the statement list is not needed. The instruction **halt** simply halts the machine. This instruction is not listed in the table.

$$\begin{aligned}
\textit{codeP}\ (\textit{class}_1;\ldots;\textit{class}_r;\ \textit{sl}) = {} & \textit{codeC class}_1; \\
& \qquad \vdots \\
& \textit{codeC class}_r; \\
& \textit{codeSL sl}\ []\ \textit{false}\ 0; \\
& \textbf{halt}
\end{aligned}$$

4.4 Methods

Certainly the most frequently used construct in O'SMALL is the sending of a message $e.m(e_1, \ldots, e_n)$. We call e the *receiver* of the message, m the *message selector*, and $e_1, \ldots, e_n$ the arguments.

$$\begin{aligned}
\textit{codeE}\ (e.m(e_1, \ldots, e_n))\ \beta \quad = \quad & \textbf{mark}; \\
& \textit{codeE}\ e\ \beta; \\
& \textit{codeE}\ e_1\ \beta; \\
& \qquad \vdots \\
& \textit{codeE}\ e_n\ \beta; \\
& \textbf{send}\ m,n
\end{aligned}$$

The first instruction **mark** creates the beginning of a new stack frame. The value of the FP register is saved into the second cell of the new frame (Fig. 5). The first cell that will contain the continuation address is left uninitialized for the moment. It will be set in the instruction **send** m,n. Now we evaluate the expression e that designates the receiver of the message. Its evaluation leaves a reference to this object (COP, current object pointer) in the third cell of the new frame (Fig. 5). Before the receiver becomes the current object, we evaluate the arguments leaving their values on the stack. The instruction **send** m,n sends the message m with n arguments to the receiver. The register FP is set to the third organizational cell containing a reference to the receiver who becomes the new current object. The continuation address is saved into the first cell of the stack frame. The method is looked up in the new current object and the register PC is set to the beginning of the code of the method. Note that the instruction **return** is the last instruction of each method's code.

$$\textit{codeE}\ \texttt{self}\ \beta \quad = \quad \textbf{pushloc}\ 0$$

The expression `self` must be translated to a code sequence that loads the current object (pointer COP) on top of the stack. The current object pointer is lying exactly where the register FP is pointing to. Thus, we can get it with **pushloc** 0.

$$
\begin{aligned}
\mathit{codeE}\,(\mathtt{super}.m(e_1,\ldots,e_n))\ \beta \quad = \quad & \textbf{mark};\\
& \textbf{pushloc}\ 0;\\
& \mathit{codeE}\ e_1\ \beta;\\
& \quad\vdots\\
& \mathit{codeE}\ e_n\ \beta;\\
& \textbf{call}\ l_m,n
\end{aligned}
$$

The sending of a message to **super** is similar to the sending of a message to **self**. We start a new stack frame. The receiver of a message to **super** is the current object, which we can get with **pushloc** 0. The arguments are evaluated as usual. The difference to ordinary message passing lies in the last instruction. The message is not looked up in the lookup table of the current object. Instead, it is directly called (static binding, not late binding). It is the corresponding method of the super class. In the instruction **call** l_m,n, the label l_m is the code address of the method and n is the number of arguments.

$$
\begin{aligned}
& \mathit{codeM}\,(\mathtt{meth}\ m(x_1,\ldots,x_n)\ \mathit{sl})\ \beta =\\
& \qquad \mathit{codeSL}\ \mathit{sl}\ \beta[x_i \rightarrow (\mathit{LOC}, i)]_{i=1}^{n}\ \mathit{true}\ n;\\
& \qquad \textbf{return}
\end{aligned}
$$

At run time, we have the following situation on the stack when the code of a method is executed. The three organizational cells of the stack frame are filled with the right values (Fig. 5) and the n arguments of the message sent are on the stack. The code for the statement list is executed leaving the resulting value on top of the stack. The instruction **return** sets the register PC to the continuation address, the register FP to the previous stack frame, and slides the result of the method upwards to the new top of the stack. As a result of all this, the current stack frame is given up and we are again in the previous one.

4.5 Classes and Objects

The code for class declarations is the most complicated one. Therefore, we will describe it in a simplified form. The exact description is contained in the code scheme.

We distinguish classes from class declarations. A *class declaration* is the syntactic object. E.g., in Fig. 1, the class `PAccount` contains the instance variable `fee` and the method `transact`. A *class* contains the instance variables and methods of all ancestor classes. In Fig. 1, the class `PAccount` has two instance variables and four (sic !) methods (one is overwritten).

$$
\begin{array}{l}
\mathit{codeC}\ (\ \texttt{class}\ c_1\ \texttt{inheritsFrom}\ c_2 \\
\qquad \texttt{def var}\ v_1 := e_1 \\
\qquad\qquad \vdots \\
\qquad\quad \texttt{var}\ v_m := e_m \\
\qquad \texttt{in meth}\ m_1(\mathit{args}_1)\ \mathit{sl}_1 \\
\qquad\qquad \vdots \\
\qquad\quad \texttt{meth}\ m_k(\mathit{args}_k)\ \mathit{sl}_k \\
\qquad \texttt{ni}\ ) = \\
\qquad\quad l_{c_1,\_init} : \mathit{codeE}\ \textbf{super.\_}\mathit{init}\ [];\\
\qquad\qquad\qquad \mathit{codeSL}\ (v_1 := e_1; \ldots ; v_m := e_m)\ [v_i \mapsto (\mathit{INST},\ \mathit{offset}+1)]_{i=1}^{m}\ \mathit{false}\ 0;\\
\qquad\qquad\qquad \textbf{return}\\
\qquad\quad l_{m_1,c_1} : \mathit{codeM}\ (\texttt{meth}\ m_1(\mathit{args}_1)\ \mathit{sl}_1)\ [v_i \rightarrow (\mathit{INST}, \mathit{offset}+i)]_{i=1}^{m}\\
\qquad\qquad\qquad \vdots \\
\qquad\quad l_{m_k,c_1} : \mathit{codeM}\ (\texttt{meth}\ m_k(\mathit{args}_k)\ \mathit{sl}_k)\ [v_i \rightarrow (\mathit{INST}, \mathit{offset}+i)]_{i=1}^{m}\\
\\
\qquad \text{where } (\_,\ \mathit{offset}) = \mathit{classInfo}\ c_2
\end{array}
$$

The code for a class consists of two parts. The first part creates and initializes new objects. This code is executed each time an object has to be created by `new c`.

$$
\begin{array}{lcl}
\mathit{codeE}\ \texttt{new}\ c\ \beta & = & \textbf{mark};\\
 & & \textbf{makeobject}\ \mathit{maketable}\ \mathit{methodSet},\ \mathit{numOfInstVars};\\
 & & \textbf{call}\ l_{c,\_init},\ 0
\end{array}
$$

where $(\mathit{methodSet},\ \mathit{numOfInstVars}) = \mathit{classInfo}\ c$

The initialization of instance variables is treated like a statically bound procedure call, i.e. like **super**. The idea behind this is a program transformation where each class is equipped with an initialization method. This method consists of calling the initialization method of the super class and a sequence of assignments to the instance variables of the current class. We write **super._*init*** for calling this initialization method. The underscore indicates that this method name is different from those that can be used in the program. We assume that the static analysis guarantees that the visibility of previously defined instance variables in later definitions. This is not complicated because O'SMALL has no global variables. The instance variables of each class declaration are initialized starting from the ancestor class declaration below class `Base` and going to the current class declaration. Class `Base` has the empty initialization code.

The second part consists of the methods of the class declaration. These methods have only access to the instance variables of this class declaration (*encapsulated instance variables*).

Before we start the translation proper we perform a phase of static analysis where the class information is collected from the class declarations, i.e. the inheritance hierarchy is unfolded. The corresponding function is called

$$\mathit{classInfo} :\ C \rightarrow 2^{\mathit{Meth}} \times \mathbb{N}_0$$

and it gives us a set of methods and the number of all instance variables of this class. The function *classInfo* is defined inductively. *classInfo* of the class `Base` is $(\emptyset, [])$. If the program contains the class definition

```
class c1 inheritsFrom c2
def var v1 := e1
        ⋮
    var vm := em
in meth m1(args1) sl1
        ⋮
   meth mk(argsk) slk
ni
```

and $classInfo(c_2) = (mset_2, numOfInstVars_2)$ then we define $classInfo(c_1)$ as

$$(mset_2 \oplus \{(m_1, c_1), \ldots, (m_k, c_k)\}, numOfInstVars_2 + m)$$

where $\oplus$ is a right preferential union operator that overwrites if there is a pair with the same method name. The pairs (m_i, c_i) of method name and class name can be regarded as the identity of a method definition. There will be a one to one mapping from these identities to labels into the code where these methods start. We assume that no class is declared twice in the program.

4.6 Explicit Wrappers

The O'SMALL dialect with explicit wrappers [7] can also be translated by our compiler. Since wrappers are top-level like classes, we can transform wrappers away by building the corresponding classes. We will not go into the details of the transformation here; the interested reader may refer to the semantics in [6, 7].

5 Conclusion

The decision to restrict O'SMALL classes to the top-level has had a great influence on the implementation of the language. The method-lookup tables can be constructed at compile time. What remains to be done at run time is to determine the class of the receiver and jump to the correct address. Therefore, late binding can be done in constant time. Of course, method inlining is impossible in the general case.

The O'SMALL-machine is considerably simpler than the MaMa or the P-Machine. This may be partly due to the simplicity of O'SMALL. But even if O'SMALL were enriched by more complicated scoping rules and additional constructs the machine would remain simple. Object-oriented programming languages like SMALLTALK and O'SMALL are simpler than e.g. Pascal because they do not have nested procedure declarations: you cannot declare a class inside another one. Programming experience shows that it is indeed not clear if nested procedure declarations are really necessary. A flat structure might well be sufficient.

The presentation of the abstract machine has been simplified in order to make it readable.

- In an implementation, one would have one method lookup table per class, not one per object as our description suggests.
- The construction of method lookup tables has not been described.
- O'SMALL is originally a dynamically typed language. The only phase where we have included a test is in the instruction **send** m,n where the method is looked up. Of course there must be further tests. In reality, every stack and heap entry must contain a tag saying whether it is a reference to an object, an integer, or a boolean. These tags have to be tested in all occasions.

There exists a prototype implementation of this machine in SML. From its level of abstraction it is between the description of this article and a fast implementation of a machine for O'SMALL.

An alternative to tags and dynamic checking would be static type checking [8]. The speedup gained by type inference and other techniques of static analysis are interesting for future research.

Acknowledgements We thank Reinhold Heckmann for carefully reading a draft version and Helmut Seidl for valuable comments.

References

[1] E. Börger and D. Rosenzweig. The WAM – definition and compiler correctness. Technical Report TR - 14/92, Dipartimento di Informatica, Universita di Pisa, 1992.

[2] C. Chambers and D. Ungar. Customization: Optimizing compiler technology for SELF, a dynamically-typed object-oriented programming language. *SIGPLAN Notices*, 24(7):146–160, 1989.

[3] C. Chambers and D. Ungar. Making pure object-oriented languages practical. In *Object-Oriented Programming Systems, Languages and Applications*, pages 1–15. ACM, Oct. 1991.

[4] C. Chambers, D. Ungar, and E. Lee. An efficient implementation of SELF, a dynamically-typed object-oriented language based on prototypes. In *Object-Oriented Programming Systems, Languages and Applications*, pages 49–70. ACM, Oct. 1989.

[5] A. Goldberg and D. Robson. *Smalltalk-80: the Language.* Addison-Wesley, 1989.

[6] A. V. Hense. Wrapper semantics of an object-oriented programming language with state. In T. Ito and A. R. Meyer, editors, *Theoretical Aspects of Computer Software*, volume 526 of *Lecture Notes in Computer Science*, pages 548–568. Springer-Verlag, Sept. 1991.

[7] A. V. Hense. Denotational semantics of an object-oriented programming language with explicit wrappers. *Formal Aspects of Computing*, 5(3):181–207, 1993.

[8] A. V. Hense. *Polymorphic Type Inference for Object-Oriented Programming Languages*. PhD thesis, Universität des Saarlandes, Fachbereich 14, D-66123 Saarbrücken, 1994. forthcoming.

[9] R. E. Johnson, J. O. Graver, and L. W. Zurawski. TS: An optimizing compiler for Smalltalk. In *Object-Oriented Programming Systems, Languages and Applications*, pages 18–26. ACM, Sept. 1988.

[10] P. Kursawe. How to invent a Prolog machine. In *Proc. Third International Conference on Logic Programming*, pages 134–148. Springer LNCS 225, 1986.

[11] D. MacQueen. Modules for Standard ML. In *Polymorphism The ML-LCF-Hope Newsletter*, 1985. Vol. II, No. 2.

[12] E. Meijer. A taxonomy of function evaluating mechanisms. In *Proceedings of the Workshop on Implementation of Lazy Functional Languages*, Sept. 1988. Report 53.

[13] U. Nilsson. Towards a Methodology for the Design of Abstract Machines for Logic Programming. *Journal of Logic Programming*, pages 163–188, 1993.

[14] S. Pemberton and M. Daniels. *Pascal Implementation: The P4 Compiler*. Ellis Horwood, 1982.

[15] R. Wilhelm and D. Maurer. *Übersetzerbau - Theorie, Konstruktion, Generierung*. Springer Verlag, Berlin Heidelberg, 1992. in German.

[16] R. Wilhelm and D. Maurer. *Compiler Design - Theory, Construction, Generation*. Addison-Wesley, 1994. to appear.