Structuur van Computerprogramma's II
Academic Year 2023-2024
The SmiLang Language

Clément Béra, Maarten Vandercammen, Elisa Gonzalez Boix
Vrije Universiteit Brussel

The goal of this session is to get a hands-on experience with call stack management. During the lectures, you saw how the stack is used during the execution of a C program. In this session, you will implement a stack machine that is able to execute program written in a minimal language called *SmiLang*. Your task will be to implement the operations necessary to execute functions in SmiLang.

This document explains the background information on SmiLang and its instruction set necessary to complete your task.

1 SmiLang Basics

SmiLang is a minimal language which can be compared to a very restricted C. It is called SmiLang because it only operates data of type integer [1]. It is said to be "minimal" because the number of operations in the language is small. In particular, it features:

- Smi integer types. We will use the term *Smi* to talk about the integers manipulated by SmiLang. For the purposes of this lab session, SmiLang will always manipulate integers of the size of a pointer (int in 32-bit and 16-bit architectures and long long in 64-bit architectures).
- Constant Smi.
- Definition of local variables, and functions.
- Access to local variables and function arguments.
- Store values to local variables.
- Calls to arithmetic operators (+ and -), user-defined functions, and `return` statement.

[1] "Smi" is the nickname for SmallInteger in several Virtual Machine (VM) teams such as Google V8 and OpenSmalltalk VM teams.

As a concrete example of a SmiLang program consider Listing 1 below. The `one` function returns the constant Smi 1. The `minus` function takes two arguments, and returns their subtraction by calling the - arithmetic operator. The `multiTempArgs` takes two arguments, defines 3 local variables and does some arithmetic operations with the arguments and local variables.

```
int one(){
  return 1;
}

int minus(int arg1, int arg2){
  return arg1 - arg2;
}

int multiTempArgs(int arg1, int arg2){
    int var1, var2, var3;
    var1 = arg1 + one();
    var2 = arg2 - arg1;
    var3 = arg2 - var1 + var2;
    return minus(var3, arg1 + 2);
}
```

Listing 1: A SmiLang program

2 A Stack-based Runtime Environment

Recall from the lectures that C features a stack-based runtime environment in which all functions are global. SmiLang works similarly but the instruction set required to generate SmiLang code is simpler than the GNU assembly one seen at class.

Let us briefly look at the compiled code generated by the given compiler for the `one` function (stored at address 0x7f8e91c02b40) shown above:

```
0x7f8e91c02b40          PushInt 1
0x7f8e91c02b49          ReturnTop 0
```

The code generated for the `one` function only consists of two instructions (stored at addresses 0x7f8e91c02b40 and 0x7f8e91c02b49): the constant 1 is pushed on the stack, and the function returns. Note that since the function does not take any argument, the `ReturnTop` instruction takes 0 as argument.

As mentioned earlier, SmiLang is a minimal language corresponding to C stripped down to only include basic stack operations to support the execution of functions. Here is the complete list of instructions of the assembly language for SmiLang:

- Instructions that don't take any arguments:

1. *Pop:* Pop top of stack.
2. *Plus:* Pop the top two values on the stack, add them, and pushes the result.
3. *Minus:* Pop the top two values on the stack, subtract them, and push the result.

- Instructions that take one byte-sized argument (called xx):

1. *PushArg:* Push the value of the xx argument on the stack.
2. *PushTemp:* Push the value of local variable xx on the stack.
3. *StoreTemp:* Store the value on top of the stack into local variable xx.
4. *DefineTemp:* Define xx local variables used in the function.
5. *ReturnTop:* Return to the caller's frame and pushes the returned value (current top of stack) to the caller's stack. The active function has xx args.

- Instructions that take one pointer-sized argument (called yy):

1. *PushInt:* Push the Smi constant yy to the stack.
2. *Call:* Call the function having yy as function pointer.

Below you can see the generated code for the `multiTempArgs` function shown above. To understand that generated code we will need to first look at the stack frame layout and the calling conventions of SmiLang.

```
0x7f8e91c02a70        DefineTemps 3
0x7f8e91c02a72        PushArg -2
0x7f8e91c02a74        Call 0x7f8e91c02b40
0x7f8e91c02a7d        Plus
0x7f8e91c02a7e        StoreTemp 0
0x7f8e91c02a80        Pop
0x7f8e91c02a81        PushArg -1
0x7f8e91c02a83        PushArg -2
0x7f8e91c02a85        Sub
0x7f8e91c02a86        StoreTemp 1
0x7f8e91c02a88        Pop
0x7f8e91c02a89        PushArg -1
0x7f8e91c02a8b        PushTemp 0
0x7f8e91c02a8d        Sub
0x7f8e91c02a8e        PushTemp 1
0x7f8e91c02a90        Plus
0x7f8e91c02a91        StoreTemp 2
0x7f8e91c02a93        Pop
0x7f8e91c02a94        PushTemp 2
0x7f8e91c02a96        PushArg -2
0x7f8e91c02a98        PushInt 2
0x7f8e91c02aa1        Plus
0x7f8e91c02aa2        Call 0x7f8e91c02c10
0x7f8e91c02aab        ReturnTop 2
```

2.1 Stack Frame Layout

As explained in the lectures, the stack holds information relative to each function activation record next to each other. Recall that for each function call the section of the stack reserved for the function is usually called a stack frame. Assuming that the stack grows upwards, the data below the stack frame is related to the caller of the function activation record and the data above the frame is related the callee.

In order to support the execution of functions, the SmiLang runtime environment needs to maintain:

- a pointer to the current function activation record: the active frame pointer (fp)
- a pointer to the top of the stack: the stack pointer (sp)
- the active instruction pointer (ip)

Figure 1 shows the information stored in a stack frame. Except for the active stack frame, i.e., the top stack frame,, a stack frame is represented in order, bottom-up, as follows:

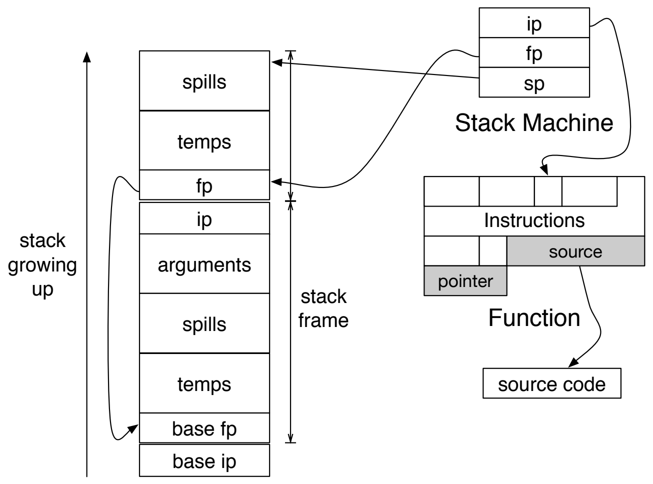

Figure 1: Abstract Stack Frame Layout

- *Local Variables:* Local variables are only accessible in a function body. In SmiLang, they are also referred as *temporary* variables. The `defineTemp` instruction reserves space on the stack for the values of the local variables of the current activation record.
- *Spilled Values:* These values are pushed on top of the stack by some instructions to be consumed by one of the next instructions. The size of this zone is variable, depending on the current execution state. It has a size

of 0 at stack frame activation time. Then, for example, if an instruction such as `PushInt` is executed, it will have a size of 1, as a Smi will be pushed on the stack.

- *Arguments:* When calling a function, the arguments are pushed on the stack. Arguments are accessed in the caller stack frame relatively to fp (the active stack frame pointer).
- *Ip and Fp:* When a function is being called, the function prologue constructs the stack frame for a new function activation record. Since the ip and fp pointers are overwritten when a new function is called, and it is important not to lose information from the previous function activation record. As such, the current values of the ip and fp pointers need to be first saved by pushing them on the stack. On return, the ip and fp values will be set back to the values that were saved on the stack.

The following figure shows a concrete representation of the stack for the `minus` function shown in Listing 1.

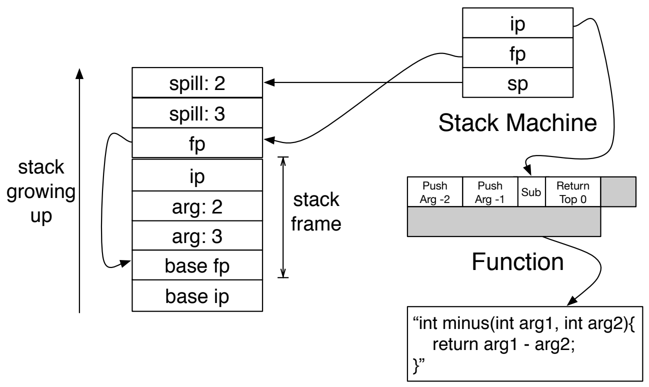

Figure 2: Concrete Stack Layout

2.2 Stack Manipulations at Work

Let us now see how the stack is used to implement simple SmiLang expressions. We start by executing the simple expression `1 + 2` step by step. This expression is compiled to the following three instructions:

1. `pushInt 1`
2. `pushInt 2`
3. `Plus`

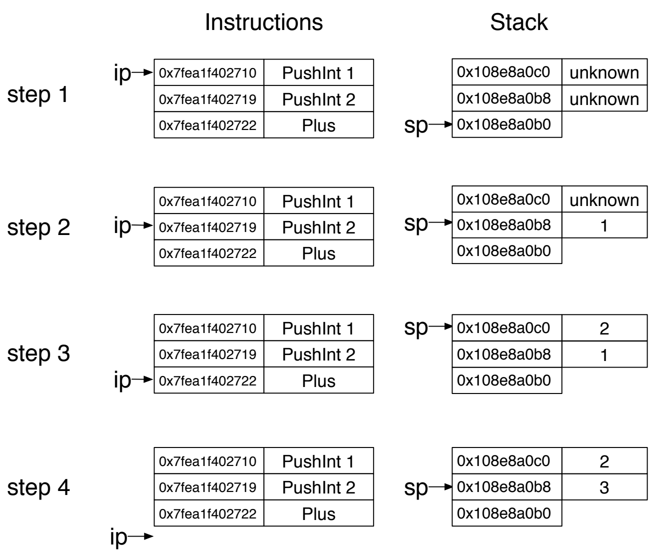

Figure 3: Execution of 1+2

Recall that the instruction pointer (ip) always points to the instruction about to be executed, and the stack pointer (sp) points to the beginning of the stack frame for the corresponding function. Figure 1 shows the step to execute `1 + 2` that we detail below:

step 1. Assume when the execution of 1 + 2 starts, the stack pointer (sp) is at 0x108e8a0b0, instruction pointer is on the first instruction and that there is enough room on the stack (2 slots).

step 2. First instruction execution (`pushInt 1`): the constant 1 is pushed on the stack. sp is now 0x108e8a0b8.

step 3. Second instruction execution (`pushInt 2`): the constant 2 is pushed on the stack. sp is now 0x108e8a0c0.

step 4. Third instruction execution (`Plus`): the two constants are popped from the stack, the result is computed and pushed on the stack, and sp is back to 0x108e8a0b8. The value at 0x108e8a0c0 on the stack is 2 but it will not be used by the execution.

Let us now execute step by step a second expression: `int var0, var1; var1 = 1; ...`. These instructions are compiled to the following instruction sequence:

1. DefineTemp 2

2. PushInt 1

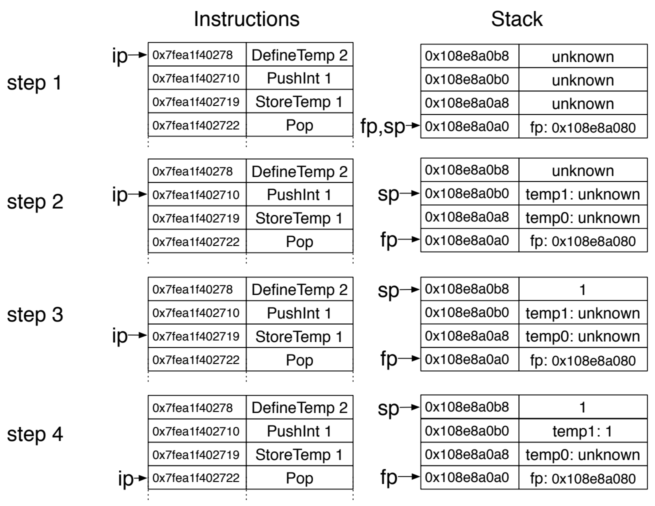

Figure 4: Execution of local variable definitions

3. StoreTemp 1

4. Pop

Figure 4 shows the step to execute `int var0, var1; var1 = 1; ...` that we detail below:

step 1. When the execution of that expression starts, the stack pointer (sp) and frame pointer (fp) point to 0x108e8a0a0. The first instruction, `DefineTemp`, moves the stack pointer to reserve space for the two local variables. The instruction encodes the number of local variables of the function, in this case 2, so the stack pointer (sp) is incremented by 16 (2 times 8 bytes, since each variable is 8 byte long) and becomes 0x108e8a0b0.

step 2. Next, the constant 1 is pushed on the stack. The stack pointer (sp) is now at 0x108e8a0b8, pointing to the value 1.

step 3. The StoreTemp instruction encodes the index of the local variable to modify, in this case, the local variable 1. The instruction stores the field on stack corresponding to that local. The address to access is computed from the frame pointer (fp) and the index of the local variable. The local variable is modified to the value on top of the stack, the Smi 1. The stack pointer (sp) remains unchanged.

step 4. At this point, the function will pop the value on top of the stack since the Smi 1 was used only to modify the local variable 1.

3 Instruction Set Implementation

Now that we understand the stack frame layout, we turn our attention to the stack machine for SmiLang. The SmiLang stack machine uses the stack to store temporary values necessary to implement each of the instructions of the SmiLang's assembly language. In this assignment we consider the stack as a block of memory addresses, and each element of the stack to be pointer-sized, i.e. 4 or 8 bytes depending on the underlying hardware architecture. In what follows we explain the SmiLang instruction set.

3.1 Pop

The `Pop` instruction pops the stack, basically decreasing the stack pointer by 1.

3.2 Plus & Subtract

`Plus` and `Subtract` pop the top two values on the stack, perform the given operation (+ or -), and push the result on the stack.

3.3 PushInt

The `PushInt` takes an integer constant as parameter, and then pushes this parameter on the stack. The stack pointer is incremented by 1.

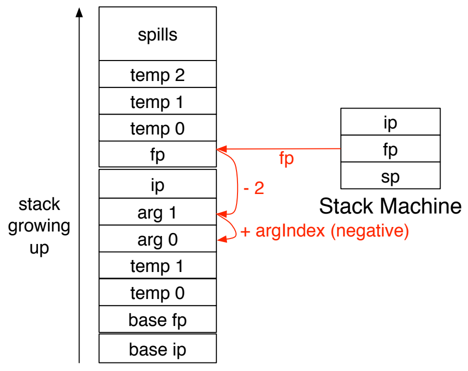

Figure 5: PushArg instruction

3.4 PushArg

`PushArg` pushes the value of one of the arguments of the function that is currently being executed to the top of the stack. Which argument is pushed is determined by the parameter of this instruction: `PushArg` -1 pushes the last argument on the stack, `PushArg` -2 the last but one argument, etc. Arguments are read by accessing the frame pointer, subtracting two to skip over the stored ip and go back to the argument zone from the frame pointer, and then subtracting the byte parameter. This is shown below in Figure 5.

3.5 DefineTemp

The `DefineTemp` instruction takes a byte-sized parameter that specifies the number of local variables used by the function. When executing this instruction the stack pointer is increased by the amount given in the parameter to reserve space for the local variables. Every local variable is also assigned the initial value `42`.

3.6 PushTemp & StoreTemp

`PushTemp` and `StoreTemp` instructions read or write a local variable. Both instructions take a byte-sized parameter, detailing the index of the local variable. `PushTemp` 0 reads the local variable at index 0. To read/write a local variable, the operation needs to access the field from the frame pointer, add one to go in the local variables zone, and add the index of the local variable. This is shown below in Figure 6.

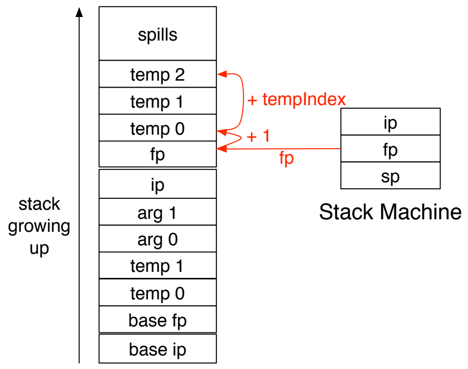

Figure 6: Local Variable Access

3.7 Call

In SmiLang, the calling convention for calls is as follows. On the `Call` instruction:

1. The pointer to the instruction where the stack machine should resume the execution of the function once the called function returns, is pushed on stack
2. The current frame pointer is pushed on stack
3. The frame pointer (fp) of the stack machine is set to the top of the stack, which by now holds the previous frame pointer

4. The instruction pointer (ip) of the stack machine is set to the first instruction of the function called (the first instruction of the function called is the address of the function itself)

All the arguments are also passed to the callee using the stack, but these arguments are not pushed via the `Call` instruction: these arguments should have already been pushed to the stack before executing the `Call` instruction. Figure 7 shows the state of the stack before and after the call.

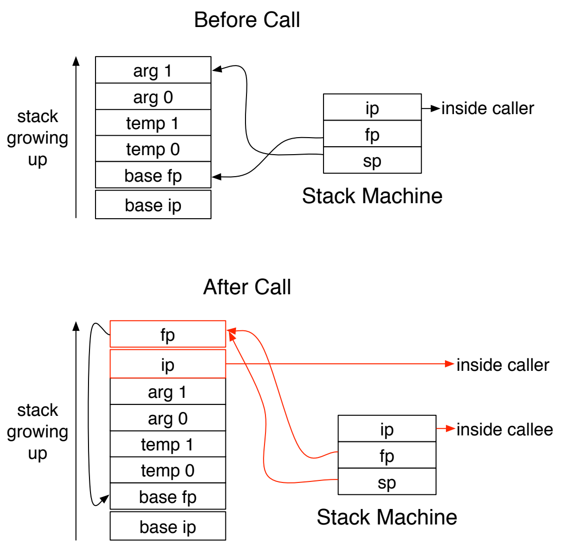

Figure 7: The Call instruction

3.8 ReturnTop

The `ReturnTop` instruction is used to return from a function, with the value that is currently on the top of the stack. In SmiLang, the calling convention for return is as follows. On the `ReturnTop` instruction:

1. The instruction pointer (ip) of the stack machine has to be restored to the instruction pointer saved just below the current frame pointer (fp)

2. The stack pointer (sp) has to be set so that the arguments of the Call instruction are consumed, *i.e.,* the stack pointer is set to the frame pointer, minus an offset for the saved instruction pointer and the current function arguments (The ReturnTop instruction precise the number of arguments of the current function)

3. The frame pointer (fp) of the stack machine has to be set back to the previous frame pointer, just by reading the value it points to

4. The returned value, computed at the beginning, has to be pushed on stack. Figure 6 shows the state of the stack before and after the return.

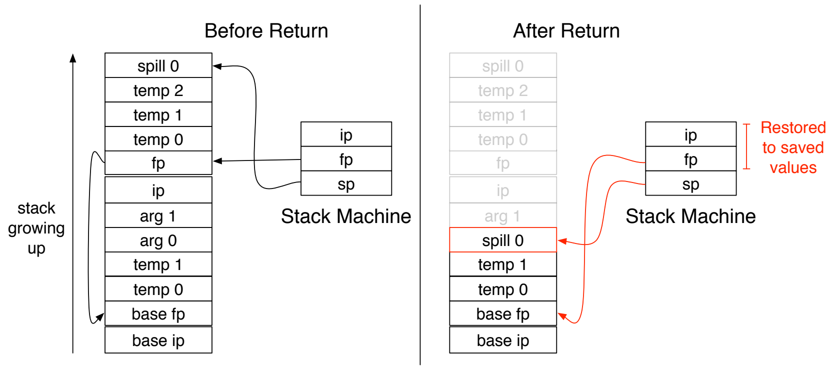

Figure 8: The ReturnTop instruction