CHAPTER 1
Introduction

This part of the book deals with call stack management, especially when the executed programming language supports closures.

To master all the concepts exposed, the reader has to implement several interpreters with an increasing level of complexity. An interpreter is a program that executes other programs written in a programming language. All the interpreters are designed to execute code written in a very simple programming language, SmiLang.

Although it is recommended to implement all the interpreters described in this part, the reader has only to understand and implement the context interpreters described in Chapters 4 and 7 and read the Chapters 2 and 10 to continue reading the book. The Chapters 5, 6, 8 and 9 deals with interpreters using a native stack and are not required to understand the next parts of the book.

Concepts

Readers will learn the following concepts during this tutorial:

- Context interpreter
- Stack Interpreter
- Stack page management
- Closure creation and activation
  - Remote temporary variable access
  - Non-local returns

Outline

This tutorial is divided in several chapters:

1. Description of the SmiLang language
2. Theoretical part: Implementations of efficent stacks
3. Implementation of a simple context interpreter
4. Implementation of a simple stack interpreter
5. Addition of stack page management in to the stack interpreter
6. Introduction of naive closures in to the context interpreter
7. Introduction of simplified closures in to the stack interpreter
8. Introduction of real closures in to the stack interpreter
9. Discussion: what are the difference between the Smalltalk model and the SmiLang model

Thanks

Thanks to Stéphane Ducasse for his contributions to this part of the book (clarifying some paragraphs and fixing typos), as well as for the early reviews and runs of the tutorial.

Thanks to Cyril Ferlicot, Christophe Demarey and Vincent Blondeau for the very early reviews and runs of the tutorial.

CHAPTER 2

Theory: Stack Implementation

This chapter describes how to implement efficiently a stack. This discussion is theoretical and does not not deal directly with call stack management. However, efficient call stack management relies on efficient stack implementations, hence call stack management is a direct application of what is described in this chapter. In fact, while explaining the different interpreters, the book refers to this chapter to detail which stack implementation was chosen for which part of the call stack and why it seemed to be the best performance-wise.

Different implementations of stacks, detailled with samples of Pharo code, are explained and discussed in the following sections. There is not such a thing as a very efficient *generic* stack implementation for the level of performance we need for an execution stack. In fact, depending on different constraints, the developer can decide to implement a stack one way or another way to reach maximum efficiency.

In high-level languages such as Pharo, people usually use `OrderedCollection` to represent different more restrictive data structures such as stacks or queues. Hence, the implementations described are usually implemented and used in low level languages, more precisely C, C++ or Rust.

The conventional APIs of a stack are `push:`, `pop` and `top`. One can consider adding a few other messages, such as `size`. However, messages such as `at:`, `do:` or `remove:` are **not** part of a conventional stack APIs.

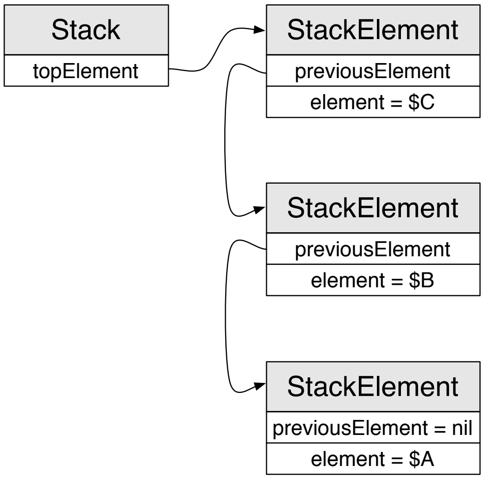

Figure 2.1 Linked List Stack where `$A, $B and $C` were successively pushed.

2.1 Naive Implementations

Linked List implementation

The first naive implementation is to represent a stack as a linked list. The stack keeps a reference to the top element, and each element keeps a reference to the previous element on stack, or nil if it's the bottom element.

An example of such a stack, where the characters `$A, $B and $C` were successively pushed, is represented on figure 2.1.

Here is possible code that could be written to implement a stack using a linked list (we omitted the getters and setters on `StackElement`).

```
Object subclass: #Stack
  instanceVariableNames: 'topElement'
  classVariableNames: ''
  package: 'Collection-Stack'
```

```
Object subclass: #StackElement
  instanceVariableNames: 'previousElement element'
  classVariableNames: ''
  package: 'Collection-Stack'
```

```
Stack >> push: value
  topElement := StackElement new
    previousElement: topElement;
    element: value;
    yourself.
  ^ value
```

```
Stack >> pop
  ^ topElement
    ifNil: [ self error: 'stack underflow' ]
    ifNotNil: [
      | value |
      value := topElement element.
      topElement := topElement previousElement.
      value ]
```

The stack implementation used in Pharo is very similar to this implementation. However, the Pharo `Stack` has many more APIs, most of them being unconventionnal for a stack.

This implementation has the big advantage that it cannot overflow (as long as there are memory available). In addition, the stack does not need to allocate memory for elements that are not yet pushed on stack.

It has however several big problems:

- each element requires an indirection to keep the reference to the previous element through an instance of `StackElement`. This has two drawbacks:
  - The memory footprint per element is increased (the stack element requires at least an extra machine word to refer to the previous element).

  - Accessing an element of the stack requires two memory reads, one to read the instance of `StackElement`, one to read the element itself.
- each element can be very far in memory from the the previous element, leading to many cpu cache miss when pushing and popping elements on stack often.

This implementation cannot be used directly for a call stack, mostly because it is dog slow. However, this design inspired better design as described in one of the following section.

Array implementation

The other naive implementation of a stack is composed of a fixed-sized array and a stackPointer. The array holds the values pushed on the stack, while stackPointer holds the index of the top of the stack.

An example of such a stack, where the characters `$A, $B and $C` were successively pushed, is represented on figure 2.2.

Here is an example of how a stack could be implemented:

```
Object subclass: #Stack
  instanceVariableNames: 'array stackPointer'
```

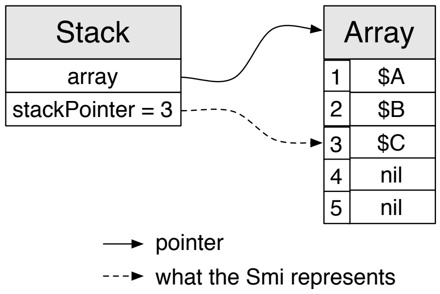

Figure 2.2 Array Stack where `$A, $B and $C` were successively pushed.

```
  classVariableNames: ''
  package: 'Collection-Stack'
```

```
Stack class >> new: size
  ^ self basicNew initializeWithSize: size
```

```
Stack class >> new
  ^ self new: 10
```

```
Stack >> initializeWithSize: size
  array := Array new: size.
  stackPointer := 0.
```

```
Stack >> push: value
  stackPointer >= array size ifTrue: [ self error: 'stack overflow'
    ].
  ^ array at: (stackPointer := stackPointer + 1) put: value
```

```
Stack >> pop
  | top |
  stackPointer <= 0 ifTrue: [ self error: 'stack underflow' ].
  top := array at: stackPointer.
  stackPointer := stackPointer - 1.
  ^ top
```

This second naive implementation have two main drawbacks:

- The stack can overflow if the programmer has not anticipated the maximum size the size can have in its lifetime, which is sometimes very difficult to anticipate.
- If the array allocated for the stack is huge whereas the stack uses during most of its lifetime only a very small portion of the array, most of the space allocated for the stack is wasted.

This implementation has advantages over the linked list implementation:

- If the average number of elements on stack is fairly close to the maximum stack size during most of the stack lifetime, then the memory consumption of this implementation is smaller than the previous implementation.

- The stack does not require an extra indirection to access its elements, providing faster access. All the element references are next to each other in memory, hence accessing successively to several elements usually ends up in cpu cache hits, greatly improving the performance.

2.2 Advanced implementations

The advanced implementation are directly inspired from the array implementation. They tackle two main problems:

- how to handle stack overflow so that the stack can dynamically grow without wasting too much memory and execution time.
- how to remove the stack overflow and underflow checks in the push and pop operations to improve their performance.

The next subsection discusses the case where stack overflow and underflow can't happen. The last two subsections deal with dynamic growth and shrink of the stack.

Fixed-sized stack

For this implementation, we assume that the operations performed on the stack are known in advance so the program can know for sure the maximum size of the stack that will used. Hence, an array of the maximum expected size can be allocated to avoid stack overflows.

Based on a simple analysis of the operations that will be performed on the stack, the program can also know before creating the stack if there will be a stack underflow or not, and raise an error before creating the stack if this is the case instead of checking at each pop operations.

The stack code or the array implementation can be simplified to remove the stack underflow and overflow checks.

```
Stack >> push: value
  ^ array at: (stackPointer := stackPointer + 1) put: value
```

```
Stack >> pop
  | top |
  top := array at: stackPointer.
  stackPointer := stackPointer - 1.
  ^ top
```

Growing the stack is useless in this implementation as the stack can't overflow.

This implementation is the most efficient possible (modulo some machine specific optimizations). However, it has some strong constraints such as anticipating the maximum size of the stack sor it cannot be used in all the cases where the programmer needs a stack.

In the rest of the book, this implementation of the stack will be referred as the **fixed-sized stack**.

Copying stack

This is the approach used by `OrderedCollection` in Pharo. Growing the stack implies the creation of a larger array with the previous array elements. Here is possible code:

```
Stack >> push: value
  stackPointer >= array size ifTrue: [ self growStack ].
  ^ array at: (stackPointer := stackPointer + 1) put: value
```

```
Stack >> growStack
  | newArray |
  newArray := Array new: array size * 2.
  array withIndexDo: [ :value :index |
    newArray at: index put: value ].
  array := newArray.
```

Once the stack has grown, it can push new elements until it overflows again (then it grows again, etc.).

This approach is reasonable but not good enough in our case. There are several problems.

Firstly, the stack grows but never shrinks, leading to huge memory footprint in some cases. Adding support for stack shrinking is difficult to do without decreasing the performance of `pop` (it would need an extra check in addition of the stack underflow check).

Secondly, the array used for the internal representation of the stack has a different size depending on the size the stack had when it overflowed the last time. Hence, it is likely that once the stack has grown, the previous array used will need to be garbage collected and can't be directly reused by another instance of stack which would need an array of the same size.

Stack pages stack

This is the stack implementation usually used for most call stacks. Initially, the stack uses a fixed-size array, called a **stack page**. When the stack page is full, which corresponds to a stack overflow in the naive array implementation described in subsection 2.1, a new stack page is allocated with a pointer back to the previous stack page and the stack keeps growing there.

In other words, the stack is represented as a linked list of stack pages. Most `push` and `pop` operations are done within a stack page so they are very efficient. From time to time, `push` and `pop` operations requires the stack to switch to another page, in a similar way to the linked list implementation of the stack discussed in subsection 2.1, which is less efficient but uncommon.

When a pop operation implies that the top of the stack is back on the previous stack page, the stack no longer uses the stack page where the popped value was, hence it can be freed. The stack is therefore successfully growing and shrinking, wasting at worst a stack page of memory, with low execution overhead.

In fact, this implementation has both the advantages of the array implementation and linked list implementation, with very little drawbacks:

- wasting at worst a stack page of memory (still better that naive implementation in most cases).
- sometimes, a `push` and `pop` operations needs to switch to another stack page which is slow.

Here follows a possible implementation.

Stack initialization:

```
Object subclass: #StackedPageStack
  instanceVariableNames: 'array stackPointer'
  classVariableNames: ''
  package: 'Collection-Stack'
```

```
StackedPageStack class >> new
  ^ self new: 10
```

```
StackedPageStack class >> new: size
  ^ self basicNew initializeWithSize: size
```

```
StackedPageStack >> initializeWithSize: size
  array := Array new: size.
  stackPointer := 1.
```

Push operation:

```
StackedPageStack >> push: value
  stackPointer >= array size ifTrue: [ self nextStackPage ].
  ^ array at: (stackPointer := stackPointer + 1) put: value
```

```
StackedPageStack >> nextStackPage
  | newArray |
  newArray := Array new: array size.
  "A reference to the previous page is kept
  to be able to pop to it later"
  newArray at: 1 put: array.
  array := newArray.
  stackPointer := 1.
```

Pop operation:

```
StackedPageStack >> pop
  | top |
  top := array at: stackPointer.
  stackPointer <= 1 ifTrue: [ ^ self fixStackPageForPop ].
  stackPointer := stackPointer - 1.
  ^ top
```

```
StackedPageStack >> fixStackPageForPop
  "The first value of each stack page is nil for the first
  stack page else a reference to the previous stack page."
  (array at: stackPointer)
    ifNil: [ self error: 'stack underflow']
    ifNotNil: [ :previousPage | self previousStackPage: previousPage
    ]
```

```
StackedPageStack >> previousStackPage: previousPage
  array := previousPage.
  stackPointer := previousPage size.
```

Example

This subsection shows some Smalltalk code pushing and popping elements on a stack implemented with stack pages aside from the memory representation of the stack. The following 6 points refer to the numbers shown on figure 2.3 and figure 2.4.

1. The stack is created and initialized with a first stack page of size 5. The program pushes on stack successively `$A`, `$B` and `$C`. Once done, the stack pointer holds 4 so the top of the stack refers to `#C` as expected. The first field of the array is `nil` to mark that this array is the first stack page so stack underflow can be detected.
2. `$D` is pushed on stack. As there is enough room on the stack page to hold `$D`, it is pushed on stack as the naive array implementation described in subsection 2.1.
3. `$E` is pushed on stack. There is not enough room on the stack page for `$E`. Hence, a new stack page is allocated for the stack. The first field of new stack page refers to the previous page so the stack will be able to pop values back to the values on the first page. `$E` is pushed on the new page.
4. `$F` is pushed on stack. As there is enough room on the stack page for `$F`, it is pushed on stack as the naive array implementation described in subsection 2.1.
5. The value `$F` is popped from the stack. After popping, there is still a value on the stack page, hence the pop operation is done as the naive array implementation described in subsection 2.1.

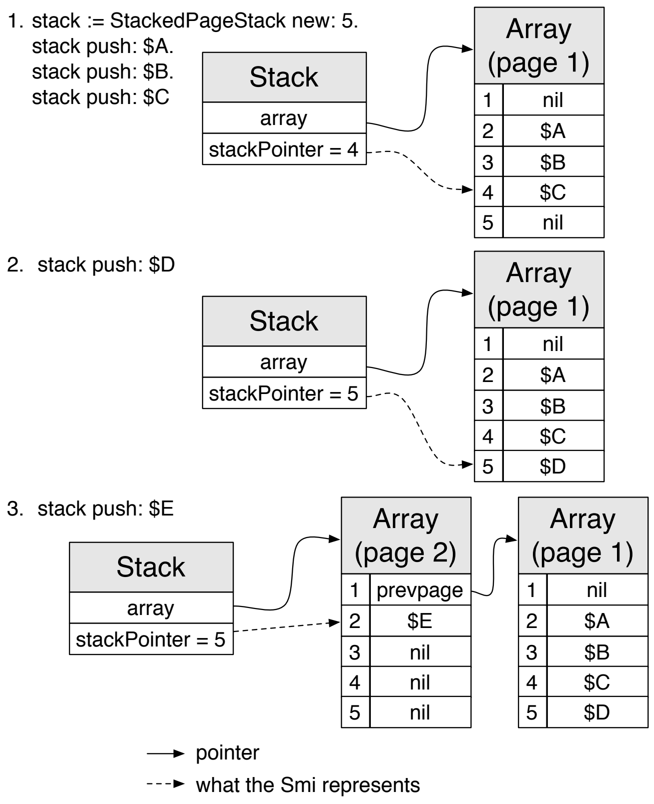

Figure 2.3 Example of successive operations on a stack with stack pages.

6. The value `$E` is popped from the stack. This time, after popping, there are no more values on the stack page. Hence, the stack page is freed. The first value of the page is used to change the reference of the stack page (`array` field) in the stack to the previous stack page. The stack pointer is updated so the top of the stack is the top of the previous stack page.

In this naive Example, stack pages of size 5 were used. It implies that 4 out of 5 `push` and `pop` operations are very efficient because they are done within a stack page, and 1 out of 5 operation is less efficient because it needs to handle the stack page switch. At worst, the stack wastes a stack page of memory, 3 fields in this case, if the last stack page is almost empty as in step 3. In addition, there are some memory overhead due each stack page needing to keep a reference to the previous stack page or `nil`.

In practice, the stack page used are far bigger (usually a few kilobytes), so the overhead of the reference to the previous stack page or `nil` is negligible, and

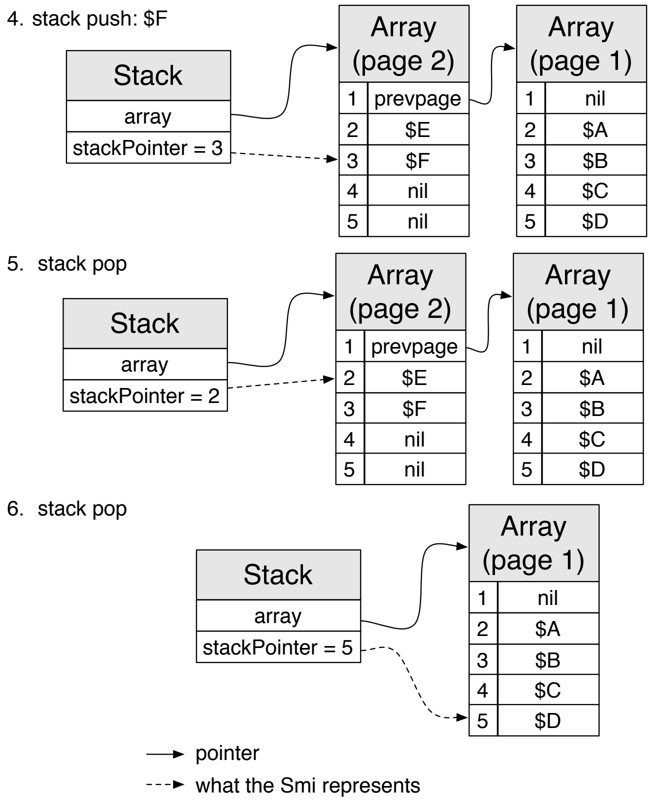

Figure 2.4 Example of successive operations on a stack with stack pages.

the stack wastes at worst a few kilobytes of memory if the last page is almost unused.

Discussion

Uncommon stack page switch

The main remaining drawback of this implementation in term of performance is the stack page switch. More precisely, it can happen for a specific stack that once initialized with enough values to fill a stack page, all its operations push one value on stack and pop it multiple times, in a way that all the `pop` and `push` operations for this specific stack are done across stack pages. This case is usually handled with extra heuristics based on the running program. It will be discussed later in this part of the book in the context of call stack management.

Unchecked pop and push operations

In the case of a call stack, the program knows ahead of time that the stack can underflow or overflow only during specific operations (function returns and function calls), hence it does not need to check for stack page underflow and overflow all the time. In fact, most `push` and `pop` operations are done within a stack page without checking if it is required to switch to another stack page, as in the fixed-sized stack implementation described in 2.2.

Reusing the stack pages

Assuming that the program uses several stacks that all use stack pages of the same size, all the stack pages used in the program have always the same size so they can be easily reused. A pool of stack pages is then available, and when a stack needs to allocate or free a stackpage, it can just request the pool to provide a page or return a page to the pool. This way, the allocation and deallocation of stack pages can be really cheap.

CHAPTER 3

SmiLang: A Simple Integer Language

During this tutorial, the reader will implement several interpreters of the language **SmiLang**. “Smi” is the nickname for SmallIntegers in several VM development teams, hence SmiLang is a language that just manipulates Smis. This chapter describes briefly SmiLang, which is a very simple language. SmiLang has only function activations, function returns, temporary variables, and some primitive operations on Smis.

3.1 SmiLang Basics

SmiLang only manipulates Smis: all the variables values are Smis and are initialized by default to 0. Functions can only be called through static calls. Every function has to return a value (no procedures).

The Smi language has 7 instructions:

1. **CallFunction**: This instruction represents a static call. The instruction has a reference to the function to call. Executing the instruction activates the function.
2. **Pop**: Executing this instruction pops the top of the stack.
3. **PrimitiveOperation**: This instruction represents a primitive operation between Smis (for example, +). Executing this instruction pops the two operands from the stack and pushes the result of the primitive operation.
4. **PushSmi**: Executing this instruction pushes a Smi constant on top of the stack. The instruction has a direct reference to the Smi to push.

5. **ReturnTop**: Executing this instruction pushes the top of stack on the context's sender or stack frame caller. Pursuing the execution resumes the execution in the calling function.
6. **PushTemp**: Executing this instruction pushes the value of a temporary variable on top of the stack. The instruction has a reference to the index of the temporary variable to push. This operation is typically used when the program needs to read the value of a temporary variable.
7. **StoreTemp**: Executing this instruction changes the value of a temporary variable for the value present on top of the stack. The instruction has a reference to the index of the temporary variable to push. This operation is typically used when the program needs to write the value of a temporary variable.

3.2 SmiLang Examples

Here are 3 examples that show the whole language:

Function one

Function `one` is a zero argument function which returns the constant Smi `1`. The operator ^ is used to represent a function return, followed by the value to return.

Source code:

```
one
  ^ 1
```

Instructions:

```
one
  <1> push 1
  <2> returnTop
```

Function add:to:

Function `add:to:` is a 2 argument function which adds the values of the 2 arguments and returns it.

Source code:

```
add: x to: y
  ^ y + x
```

Instructions:

```
add:to:
  <1> pushTemp: 2
  <2> pushTemp: 1
  <3> primitive: +
  <4> returnTop
```

Function example07TempAndCallAndCallArgs

Function `example07TempAndCallAndCallArgs` is a zero argument function with two static function calls to the functions named `one` and `add:to:`. Each call is written using the keyword `call` followed by the keywords of the function named to call separated by the arguments values. The function also uses a temporary variable, named `t` and declared at the beginning of the function with `| t |`. The operator `:=` is used to represent an assignment.

Source code:

```
example07TempAndCallAndCallArgs
  | t |
  t := call one.
  ^ call add: t to: 2
```

Instructions:

```
example07TempAndCallAndCallArgs
  <1> call one
  <2> storeTemp: 1
  <3> pop
  <4> pushTemp: 1
  <5> push 2
  <6> call add:to:
  <7> returnTop
```

We are set

The complete SmiLang has been introduced. This language may not be usable for a real application but it is perfect for our exercise which consists of the different implementations of function calls and returns.

CHAPTER 4

Context Interpreter

A context interpreter is an interpreter which represents the activation of each function call by a data structure called a context. The call stack is composed of a linked list of contexts.

4.1 What is a Context ?

A **context** is a data structure which represents a function activation. In this chapter, contexts are instances of the class `SmiContext`.

A context holds information such as:

- The values of local variables.
- The program counter (also called instruction pointer).
- The function being executed (because the function holds the instructions to be executed).
- The context that activated it (also called caller or sender).

Figure 4.1 presents a given computation state showing the relationships between context, stack, function, and instructions in SmiLang.

Let us explain each data structure present in this Figure.

Context.

- The `sender` field of the context references another context, the one which activated it. Such context chain is created during function call: the sender context has temporarily stopped its execution just after the function call that activated the described context.

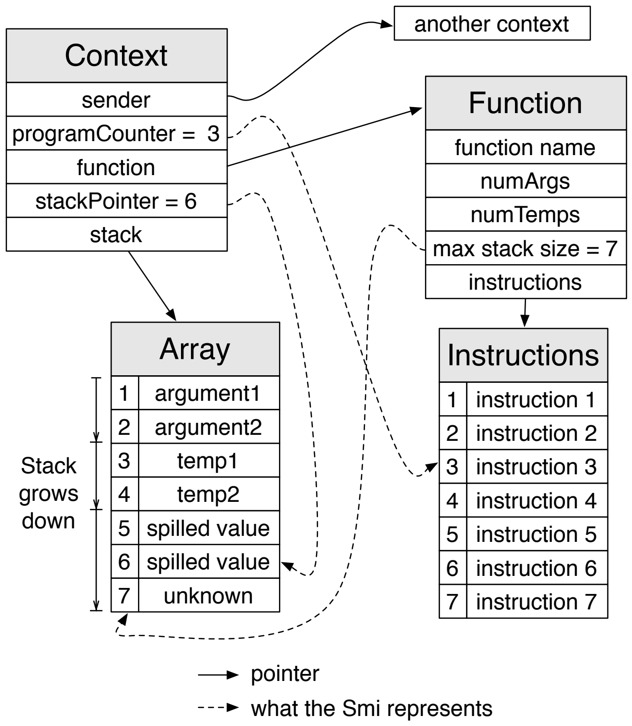

Figure 4.1 Relationships between Context, Stack, Function and Instructions in a context interpreter.

- The `program counter` field holds 3, which means the next instruction the interpreter will execute is the third instruction of the function.
- The `function` field is a reference to the function being activated. A context refers to the function because the function is a holder of instructions to be executed.
- The `stack pointer` field holds 6, which means that in the space allocated for this specific context stack, which is in our case 7, only 6 values are currently used and the value at position 6 is the top of the stack.
- Lastly, the `stack` field references the array internal to the stack representation. The context's stack values are specific to the function activation and independent from other contexts.

Function.

- The field `name` is useful for debugging purpose.

- The fields `numArguments` and `numTemporaries` describe the information mandatory to the execution of the function. They are determined by the compiler. The interpreter and debugger use these values to figure out, for each value on stack, if they correspond to an argument, a temp or a spilled value (value temporarily pushed on stack for the execution of instructions, detailled later in the chapter).
- `maxStackSize` . For performance reason, the context, on creation, allocates for its stack enough space for the maximum size that its stack can take during the function execution. This maximum size is specified by the compiler. In Figure 4.1 the field `maxStackSize` holds `7`.

Stack.

A stack is associated to each context, i.e., each function activation. A context's stack is composed by the following values, in this specific order:

1. The values of the function arguments.
2. The values of the temporary variables.
3. The spilled values, which are the values pushed on stack by previously executed instructions.

The compiler can guarantee, for a given function, that the stack of its activation does not need to be bigger than a size it computes (`maxStackSize` field of a function) and that the instructions generated will not lead to stack underflows or overflows at execution time. Hence, the context can use the *fixed-size stack* design described in the subsection 2.2. Instead of creating a separate data structure (i.e., the context refering to the stack, the stack itself refering to its internal `array` and `stackPointer`), to limit the number of memory reads, the context directly holds the `stackPointer` and the array required by the stack representation (this array is named `stack` in the implementation you will work on). On figure 4.1, the context have the fields `stackpointer` and `stack`, the stack being a reference to an array.

4.2 Concrete Example: Activation of function add:to:

Remember `add:to:` function definition is:

```
add: x to: y
  ^ y + x
```

This function is invoked in particular by the function `example07TempAndCallAndCallArgs`.

```
example07TempAndCallAndCallArgs
  | t |
  t := call one.
  ^ call add: t to: 2
```

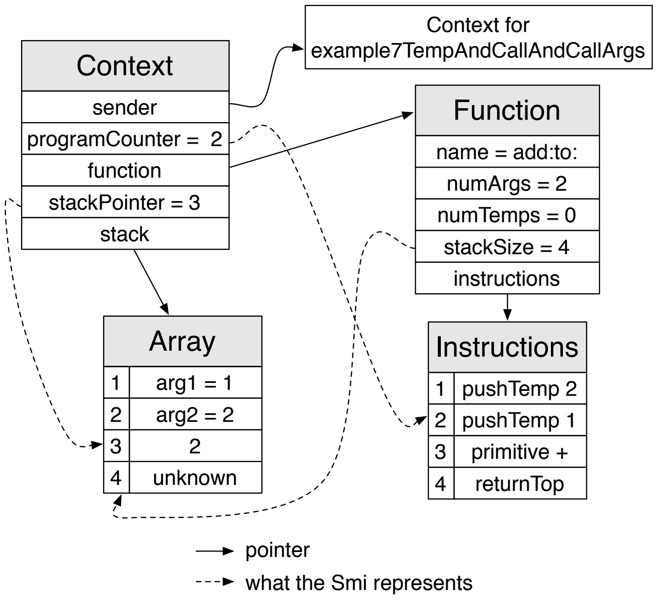

Figure 4.2 Context and collaborators for the `add:to:` call in `example07TempAndCallAndCallArgs`.

In Figure 4.2, the context represents the activation of the function `add:to:` within the function `example07TempAndCallAndCallArgs`.

The sender field holds the activation of the function `example07TempAndCallAndCallArgs`. As we can see, the function has a call to the function `add:to:`.

The programCounter is 2, which means that

- The first instructions `1| pushTemp 2` has already been executed or is currently being executed.
- The second instruction, `2| pushTemp 1` is the next instruction to be executed. Note that in most interpreter implementations, the `pushTemp` instruction works both for arguments and temporary variables depending on the value of the temp index encoded in the instruction. For store, it reads the value to store on the stack but does not change the stack pointer. For push, it pushes the value of the temporary variable on the stack. In this case the instruction is used to access the arguments values.

The function field holds a reference to the function `add:to:`. This function has 2 arguments, 0 temporary variables and a stack that can go up to 4 slots. It has a list of 4 instructions.

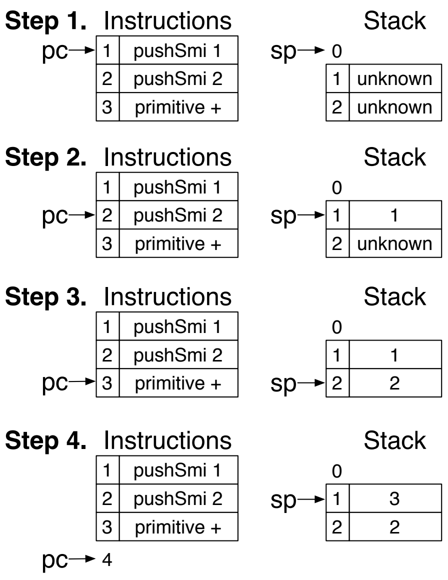

Figure 4.3 Stack state at each pc during the interpretation of `1 + 2`.

The current stack holds the values of the 2 arguments and the spilled value 2 due to the first instruction `1| pushTemp 2` that has been executed.

4.3 Interpretation

Now let us look at two examples of execution.

Example 1: Interpretation of `1 + 2`

Figure 4.3 shows the state of a context's stack at each pc during the execution of 1 + 2.

- **Step 1**: When the interpretation of 1 + 2 starts, stackPointer is at 0, 2 slots are already allocated on stack for the computation, pc is 1.
- **Step 2**: First instruction execution (pushSmi 1): the constant 1 is pushed on stack. stackPointer is now 1.
- **Step 3**: Second instruction execution (pushSmi 2): the constant 2 is pushed on stack. stackPointer is now 2.
- **Step 4**: Third instruction execution (primitive +): the two constants are popped from the stack, the interpreter computes the result (by asking the host language or the cpu) , and pushes the result on stack.

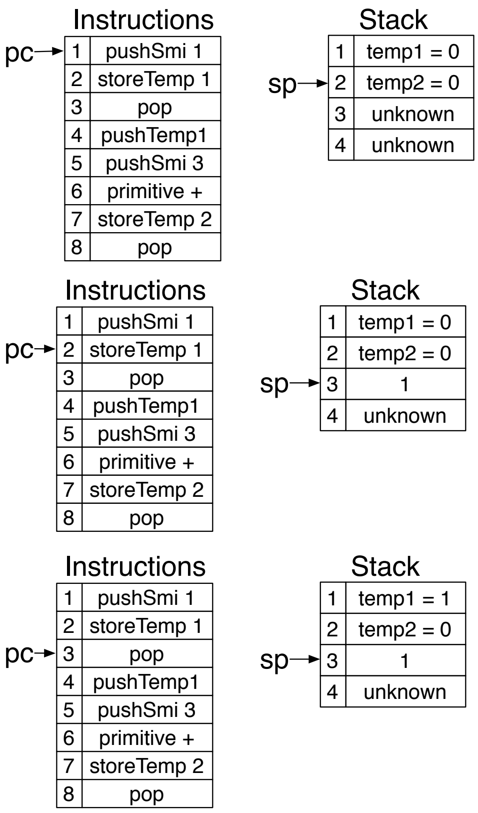

Figure 4.4 Stack state for the first pcs during the interpretation of `temp1 := 1. temp2 := temp1 + 3.`

stackPointer is now 1. The value at the second stack position is 2 but it should not be reached by execution any more: it is what is called a spilled value.

Example 2: Interpretation of `temp1 := 1. temp2 := temp1 + 3.`

The first steps of the interpretation are drawn in Figure 4.4. Try to draw or imagine the next steps.

4.4 Calls and Returns

In the two first examples we described the instructions that work inside a function activation. Now let's discuss the two interesting instructions, function calls and returns. They require the creation or the termination of a function activation.

Function calls

The call instruction creates a new context, pops the arguments from the top of the stack and copy them to the new context's stack. It then proceeds execution in the freshly created context. Indeed, remember that prior to the call execution, the argument values are in top of the stack of the current context. Once the call is actually performed, the arguments are the first elements of the stack followed by the temporaries of the new function.

Returns

The returnTop instruction pushes the top of the stack of the active context on the top of stack of the sender context. It then terminates the active context and resumes execution in the sender context.

4.5 Build a Context Interpreter

In this exercise, you will build a context Interpreter.

Open a Pharo image, load the package **SmiLang** you can find on http://smalltalkhub.com/mc/ClementBera/3DaysVM/main.

You can use the following repository description:

```
MCSmalltalkhubRepository
  owner: 'ClementBera'
  project: '3DaysVM'
  user: ''
  password: ''
```

The package SmiLang contains several tags each one grouping the classes corresponding to different aspects of SmiLang: compiler, context interpreter, stack interpreter, and tests.

For now let us focus on the context interpreter and its companion test class `SmiContextInterpreterTests`. A class `SmiContext` is given as a possible implementation of a Context for SmiLang. A `SmiContextInterpreter` class skeleton is given, but many methods are not implemented (they only hold `self shouldBeImplemented`). You will have to define them.

The interpretation starts with a fake Context which holds flags. The interpreter then calls a function (the public API, `interpret:`, specified in the public method protocol, specifies the function to call). The interpreter keeps interpreting instructions until it returns to the flagged context, which signals the end of the execution. It then returns the value computed.

The class SmiContextInterpreterTests defines a set of tests that have to pass. It is possible to look at the code and the compiled version of the examples.

Here is the code of the first test, where you see the definition and instructions of the method under test and the actual test.

```
testExample1
  "This test ensures that the execution of the function
    example01Return is correct.
  -- Source code --
  example01Return
  ^ 1
  -- Instructions --
  <1> push 1
  <2> returnTop"
  self
    assert: (self runSmiFunction: (self compilePharoMethod:
    SmiExamples>>#example01Return))
    equals: 1
```

Good luck and have fun. In the next chapter you will implement an interpreter using a native stack.

CHAPTER 5

Simple Stack Interpreter

In this chapter, one has to build a stack interpreter. Instead of considering contexts as separated objects, this interpreter uses a contiguous memory space where function activations reuse arguments pushed on the stack instead of copying them over the new function activation.

The interpreter is considered as simple as it conceptually holds a single monolithic **infinite stack**. The next chapter of the book explains how to handle a finite stack. The context interpreter, as implemented in the previous chapter, has several issues:

- Time is wasted when copying the arguments at each function call from the caller context to the freshly created context. If one would design a context interpreter which does not copy the arguments, then a context needs access the arguments through the caller context. As the caller context may be far in memory (hence, maybe not on the cpu cache), it is in practice even slower.
- A second problem is that Contexts are separate data structures and hence can be allocated by the host language, like C, far from each other. This implies that memory reads are spread in the memory and this is problematic for performance (mismatch with cpu caches).
- Another issue is that hardware-implemented cpu instructions like calls and returns are difficult or even impossible to use directly with contexts when in the future the developer will add a just-in-time compiler.

5.1 Call Stack

In one sentence, a stack interpreter is a context interpreter where all the contexts have been merged into a single stack called ”call stack” or ”native

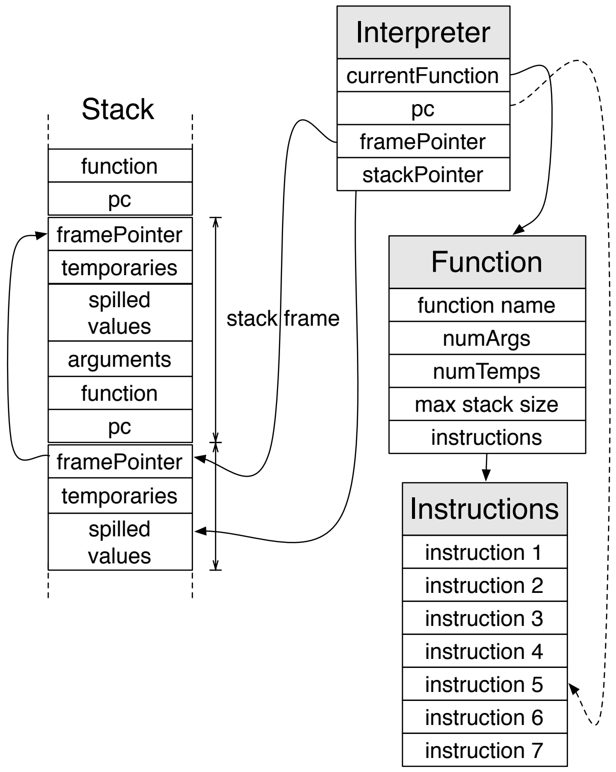

Figure 5.1 Representation of a native stack.

stack”. As the stack is infinite and the first stack frame is tagged to mark the beginning of the stack, the interpreter does not need to check for stack underflow or overflow anywhere. The implementation of the stack is the fixed-array stack described in 2.2, with an array that has conceptually an infinite size and in practice a size bigger that what the interpreter needs in the worst case. The interpreter will hold directly references to the stack pointer and the array holding the stack entries.

Call stack representation

The figure 5.1 describes the representation of a native stack.

The native stack holds information relative to each context next to each other. Assuming that the stack grows down, the data up to a stack frame is related to the sender of the function activation and the data down is related the function called by the activation.

Interpreter state

The interpreter, instead of knowing the active context, knows directly the state of the active function activation:

- the active function.
- the pc (program counter) being executed.
- the active frame pointer: it points to the beginning of the function activation in the native stack.
- the stack pointer, it points to the top of the native stack and the top of the active stack frame (they are the same). Values will be pushed and popped there.

Stack frame representation

In the native stack, we call **stack frame** a portion of the stack that represents a function activation (it corresponds to a context in the previous interpreter, except that the stack frame size is variable depending on how many values were pushed on stack before the activation of the next stack frame). Except the active stack frame, a stack frame is represented in order as follow:

- **Temporaries**: Allocated at stack frame creation, this space hold the values of the temporary variables of the activation. The size of this space is known thanks to the `numTemps` property of the function.
- **Spilled Values**: These values are pushed on top of the stack by some instructions to be consumed by one of the next instruction. 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 a `pushSmi` instruction is performed, it will have a size of `1`, as the Smi will be pushed on stack.
- **Arguments**: When calling a function, the interpreter pushes the arguments of the function called on the stack. On the contrary to the context interpreter, the arguments are never copied. Arguments are accessed in the caller stack frame relatively to the active stack frame frame pointer (this is detailed later in the chapter).
- **`Function`, `pc` and `frame pointer`**: Once a new function is activated, the interpreter can't hold anymore the pc, the function and the frame pointer of the previous activation. It needs to hold the information related to the new stack frame created for the function being called. To remember the information, the interpreter pushes on stack the active function, pc and frame pointer. On return, the interpreter will set its values with the values on stack. The framePointer holds a reference to the beginning of the previous stack frame.

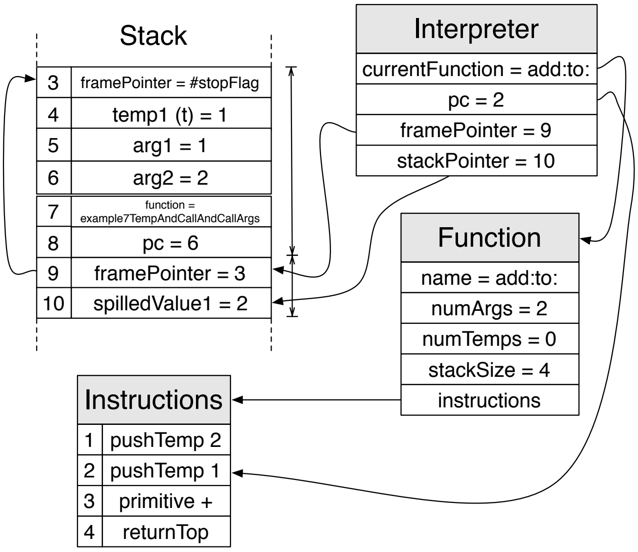

Figure 5.2 Example of a native stack.

Active stack frame

The active stack frame, i.e. the bottom stack frame, is different. The active function, pc and framepointer are directly held by the interpreter (they are not on stack). In practice, this avoids memory reads and writes if the interpreter variables (currentFunction, pc, framePointer) are in registers instead of in memory.

Concrete example: Activation of the function add:to:

The figure 5.2 describes the same example than for the context interpreter in Section 2. Don't hesitate to compare both architectures if you don't understand everything.

```
add: x to: y
  ^ y + x
```

The interpreter holds:

- A reference to the function being executed, the function `add:to:`.
- The pc of the next instruction to execute, 2.
- The frame pointer corresponding to the beginning of the current stack frame for the activation of the method `add:to:`.
- The stack pointer, which is the last value being used on stack and the top of the active stack frame.

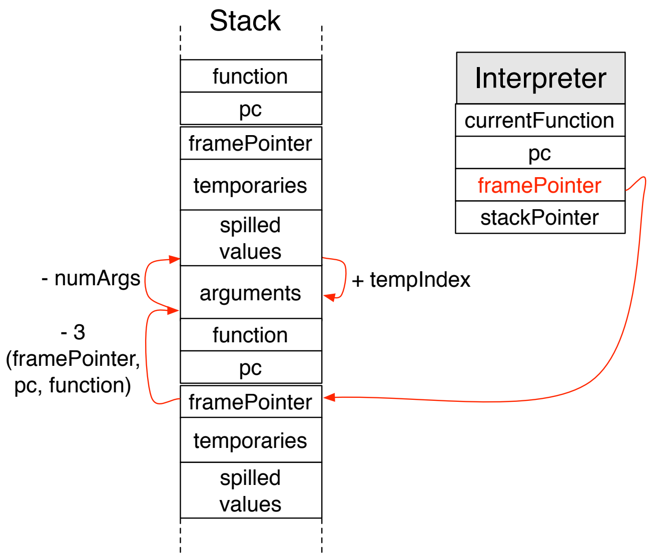

Figure 5.3 Accessing the value of an argument.

The active stack frame starts with a framePointer which refers to the frame-Pointer of the previous stack frame, i.e., the beginning of the previous stack frame. For example, in figure 5.2 the interpreter frame pointer points to slot 9 which refers to the previous stack frame start.

5.2 Temporary access

The interpreter assumes that the compiler forbids writing into the function arguments (compilation error). Hence, the store into temporary variable instruction remains simple as it assumes the temp index is the index of a temporary variable. Remember that the temporaries are now stored at the beginning of the stack frame (just after the frame pointer).

The `pushTemp` instruction is different. Depending on the tempIndex and the value of numArgs in the function, two cases are found:

- **The tempIndex corresponds to an argument.** Its value is accessed in the caller frame relatively to the active frame pointer. The previous frame has at its end the argument values, its function and its pc. The interpreter needs to substract 3 from the stack pointer to reach the zone in the previous stack frame holding arguments, then substract the number of argument and add the tempIndex to access the correct field, as shown on figure 5.3.

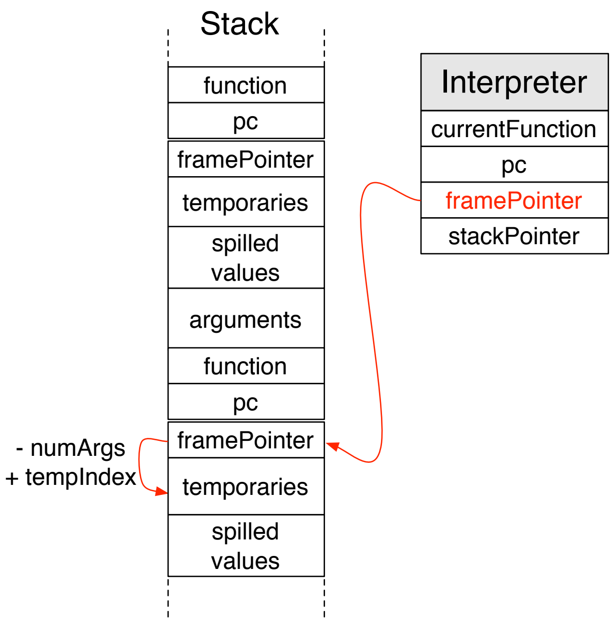

Figure 5.4 Accessing the value of a temporary variable.

- **The tempIndex corresponds to a temporary variable.** Hence we can access the value relatively to the framepointer. We need to add the tempIndex to the framePointer and substract the number of arguments as they are stored on the caller frame, as shown on figure 5.4.

5.3 Call Instruction

The call instruction is different from the context interpreter as one needs to update the interpreter state and create a new stack frame instead of simply creating a new context.

The call instruction works as follow:

- Pushes the pc, the active function and the frame pointer on stack.
- Sets up the stack zone after the new frame pointer with only slots for temporary variables (no slots for arguments as they are accessed from the caller frame). It usually implies to update the stackPointer and set the temporary variable fields to a default value (0 in our case).

5.4 Return instructions

The return instruction has to set the execution to the previous stack frame execution state. This implies popping the stack up to the previous frame pointer, then popping and restoring the previous active function, frame pointer and pc, and lastly popping the arguments of the function call. In addition, it needs to push the returned result on stack.

5.5 Build a stack interpreter

In this exercise you will build a Stack Interpreter.

Open a Pharo 4 image, load the package `SmiLang` you can find on http://smalltalkhub.com/mc/ClementBera/3DaysVM/main.

Go to the package tag `SmiLang>>Stack interpreter`. A `SmiStackInterpreter` class skeleton is given, but many methods are not implemented (they only hold `self shouldBeImplemented`).

The interpretation starts with a fake stack frame which holds flags. The Interpreter then calls a function (the public API, `#interpret:`, specified in the public method protocol, defines the function to call). The interpreter keeps interpreting instructions until it returns to the flagged stack frame, which signals the end of the execution. It then returns the computed value.

The `SmiStackInterpreterTests` have to pass to proceed to the next chapter.

Good luck and have fun !

CHAPTER 6

Stack pages

6.1 Problem

The Stack interpreter built in the previous chapter is very nice as it introduced many concepts. It has however a critical flaw: the stack is considered as infinite. An infinite stack is possible theoretically but not in practice as you need to specify to the computer a size to allocate. This size can't be too big because if the program uses multiple processes, each process requires a separate stack.

6.2 Solution

A common solution is to split the stack into stack pages of limited fixed size, as described in the subsection 2.2. When the stack page overflows, the interpreter switches to a new stack page. All the stack pages are created at interpreter start-up and there are a fixed limited number of stack pages available. The interpreter never allocates new stack pages or free them, it instead reuse the pre-allocated stack pages. Once all the pages are full, different strategies exist depending on the language you want to implement:

- Raise a StackOverflow error.
- Serialize a few stack page to free them, and once a serialized stack page needs to be used again materialize it.

To avoid checking at each `push` instruction if the stack page overflows or not, the interpreter uses the `maxStackSize` field of the function executed to predict at stack frame creation time if the stack is going to overflow or not during the function execution. If a stack overflow is going to happen, the interpreter switches to a new stack page. There is no such thing as a stack frame split between two pages, it cannot happen.

On the contrary to the simple stack with stack pages described in subsection 2.2, each stack page can have several unused fields. Indeed, the interpreter switches to a new stack page if there is not enough room for the next function execution, even if there are a few free fields left on the active stack page.

This complexifies a bit the stack page management, as when popping across stack pages, the stack pointer needs to be restored to the correct field in the previous stack page which is not necessarily the top field of the page. To know what value the stack pointer has to be restored to, the stack has to keep additionnal metadata at the beginning of each page. The simple scheme described in subsection 2.2 only keeps the reference to the previous stack page at the beginning of each page, whereas this implementation needs to keep in addition the stack pointer value to restore.

Implementation

The implementation is as follow:

1. **Function Activation**: In addition to what is done in the `StackInterpreter`, the interpreter estimates the stack frame size based on the `maxStackSize` field of the function to activate. If there is not enough room in the stack page for the next stack frame, the interpreter switches to a new stack page.
2. **Return**: In addition to the `StackInterpreter` behavior, if the stack frame returning is at the top of a stack page, the interpreter returns to a stack frame in another stack page. The top of a stack page has to be marked (for example, the return pc of the first stack frame of a page can be a flag, with the actual pc in the stack page metadata first fields) so the interpreter knows it has to do a return across stack pages.

6.3 Exercise

As an exercise, have the `SmiStackInterpreterWithStackPagesTests` pass by implementing the `SmiStackInterpreterWithStackPages`. You only need to reimplement function calls and returns to fit the new behavior described in the previous subsection compared to the interpreter of the previous chapter.

6.4 Making Calls and Returns across Stack Pages Uncommon

A good `StackInterpreter` needs to be implemented in a way that returns and calls across stack pages are uncommon because they are slower than regular calls and returns. Typically, in the case of recursive fibonnacci, it could be that most of the sends are done across stack pages and the system become dog slow.

To avoid this problem, a good stack interpreter keeps a reference to the last stack page that overflowed and the number of time it overflowed. Each time a stack page overflows, if it is the last one that overflowed, the number of overflows is incremented by 1, else the reference to the last stack page that overflowed is updated and the number of overflow reset. In pseudo-code, that would mean:

```
stackPageOverflow: stackPage
  lastStackPageThatOverflowed = stackPage
    ifTrue: [ numOverflow := numOverflow + 1 ]
    ifFalse: [
      lastStackPageThatOverflowed := stackPage.
      numOverflow := 1 ]
```

When a new stack page is required, if the current stack page has already overflowed, the interpreter decides to copy a certain number of stack frame to new page to make call and return across stack pages less common. The number of stack frame to copy depends on the number of time the stack page has overflowed and is bounded by a maximum value. This cheap strategy makes calls and returns across stack pages uncommon.

CHAPTER 7

Naive closures in the Context Interpreter

7.1 Introduction

In this chapter, the reader will introduce closures the context interpreter. This chapter deals with the closure implementation and does not explain what is a closure. If you don't know what is a closure, please read first documentation available on the internet about it, such as the chapter **14. Blocks: a Detailed Analysis** of the free online book **Deep into Pharo**.

A closure is an object or a data structure that references a function and its enclosing environment. In this implementation, the function is anonymous and the enclosing environment is the context in which the closure was created, that we call the closure's outer context, as represented on figure 7.1.

In addition to regular function, closure's function activation (contexts representing closure activations) have a reference to the closure they activated, as emphasized on figure 7.2.

If the closure is defined in another closure, then the closure's outer context is a closure context, which has itself a closure with an outer context. The list of outer context from the closure's direct outer context to the first context which is not a closure activation is called **outer context chain**. Such an outer context chain is represented on figure 7.3.The last context of the chain is called the closure's **home context**. If the closure is not nested in another closure, then the closure outer context is its home context.

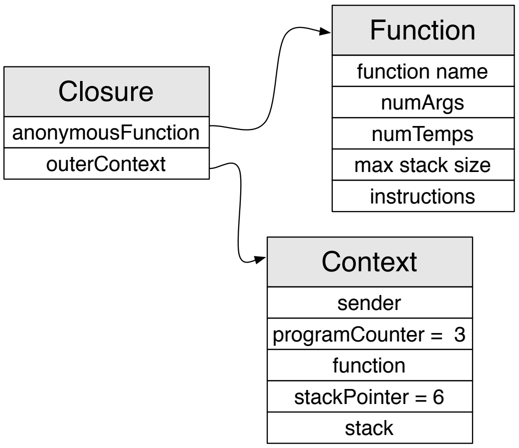

Figure 7.1 Closure representation.

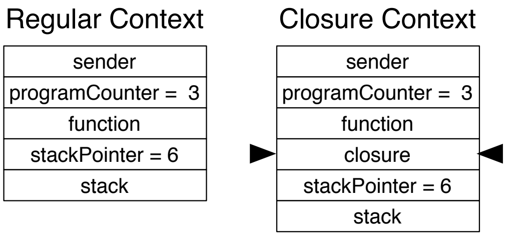

Figure 7.2 Context versus Closure Context.

Extending the language

To support closures, the syntax of SmiLang is extended with the operator `[ ]` and the primitive operation `value`.

Closure creation

`[ ]` is used to create a new closure. `[` mark the beginning and `]` mark the end of the anonymous fonction held by the closure.

Example

In the following example, the function answers a closure holding the anonymous function answering `1`. Each execution of the function answers a different closure.

Source code:

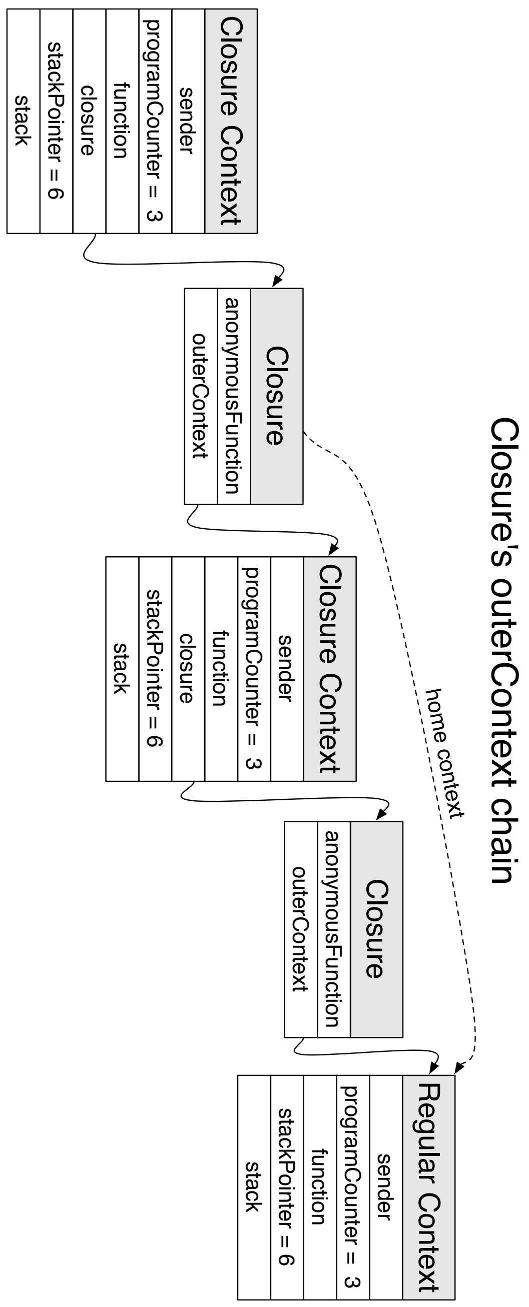

Figure 7.3 Outer Context chain.

```
example01Closure
  ^ [ 1 ]
```

Instructions:

```
example01Closure
  <1> createClosure
  <2> returnTop
```

Closure Instructions:

```
noname
  <1> push 1
  <2> returnTop
```

Closure activation

The keyword `value` represents the primitive operation that activates the closure. In a similar way to the function call, `value` creates a new context, but the context is a closure's activation and not a simple function activation. Once activated, the closure's anonymous function is executed and its result is returned.

Example

In the following example, the `value` primitive operation in the function activates the closure holding the anonymous function answering `1`, hence the result of the function is `1`.

Source code:

```
example02ClosureActivation
  | temp |
  temp := [ 1 ].
  ^ temp value
```

Instructions:

```
example02ClosureActivation
  <1> createClosure
  <2> storeTemp: 1
  <3> pop
  <4> pushTemp: 1
  <5> activateClosure
  <6> returnTop
```

Closure Instructions:

```
noname
  <1> push 1
  <2> returnTop
```

Values of temporary variables

As `[ ]` answers a closure, it is now possible for temporary variable to hold either a Smi or a closure.

We will consider that it is the developer responsability to call the primitive `value` on a closure and not on a Smi.

If `value` is called on a Smi or if a primitive operation on Smis such as `+` is called with a closure as operand, the interpreter raises an error and stops executing code.

In a real system, different solutions exists to avoid such runtime errors. For example, a type system allows the compiler to raise a compilation error if a type missmatch is detected. Alternatively, in dynamically-typed languages like Smalltalk, the runtime executes a fall-back routine when a primitive operation fails to handle such problems.

Extra features of closure activations

On the contrary to regular context representing functions, context representing a closure activation can use the closure's outer context for two different purposes:

- Remote temporary access
- Non local return

Remote temporary access

The first feature, present in most languages having closures, is the access of remote temporary variables. The closure's function can access temporary variables that are held not in the current context, but in any of the context in the closure's outer context chain.

Here is an example of a closure accessing a remote temporary variable (the instructions are detailled later in the chapter):

```
example04ClosureRemoteTemp
  | remoteTemp |
  remoteTemp := 1.
  ^ [ remoteTemp ] value
```

As you can see, the context representing the activation of the function `example04ClosureRemoteTemp` holds the value of `remoteTemp`, and the context representing the activation of the closure `[ remoteTemp ]` can access its value.

Non local return

The second feature, present at least in Ruby in the form of Procedure returns and in Smalltalk, is called non local return. When a closure's activation performs a non local return, it returns from the closure's home context instead of the closure's context sender. To learn more about non local returns, you can read the section **14.4 Returning from inside a Block** from the free online book **Deep into Pharo**.

Let's take two examples:

Source code:

```
exampleLocalReturn
  | temp |
  temp := [ 25 ] value.
  ^ temp + 1
```

Closure instructions:

```
noname
  <1> push 25
  <2> returnTop
```

Source code:

```
exampleNonLocalReturn
  | temp |
  temp := [ ^ 25 ] value.
  ^ temp + 1
```

Closure instructions:

```
noname
  <1> push 25
  <2> nonLocalReturnTop
```

In the first case, `exampleLocalReturn`, the closure has a local return. Upon execution, the temporary temp is assigned the value `25`, then the function answers `26` (`25 + 1`).

In the second case, `exampleNonLocalReturn`, the closure has a non local return. Upon execution, the closure does not return to the temporary assignment, but returns directly to the sender of the function `exampleNonLocalReturn`, so the function answers `25`.

As shown, in closures, ^ is used for non local returns whereas if nothing is precised the last value of the closure is returned locally.

7.2 Extending the instruction set

To use closures in the context interpreter, we extend SmiLang with 5 new instructions.

- **ActivateClosure**

Executing this instruction pops the closure on top of the stack and activates it. If there is something else than a closure on top of the stack, the interpreter raises an error and stop the execution of the program.

This operation is very similar to a function call as it creates a new context for the closure's anonymous function. However, the closure activation context needs a reference to the closure in addition to all the information a regular context has. Hence, closure's activations are represented as a separate data structure (in our case, a separate class) as they need an extra field.

- **CreateClosure**

This instruction represents a closure creation. Executing this instruction creates a new closure and pushes it on top of the stack. The closure is created with the anonymous function of the instruction and the context that created the closure (the active context at closure creation time).

- **NonLocalReturnTop**

Executing this instruction returns the execution to the closure's home context sender instead of the active context sender. If the closure's home context is already terminated, the interpreter raise a BlockCannotReturn exception.

When performing a non local return, all the contexts in-between the closure activation and the closure's home context sender need to be terminated. This way, other closures with other home context will always know if their home context is terminated or not.

There are different ways to represent a terminated context. One possibility is to represent a terminated context as a context which pc is nilled out. This implies nilling out the pc field of the context upon return and nilling out the pc feld of several contexts in case of non local return.

- **PushRemoteTemp**

This instruction holds the index of the temporary variable to access but also the number of outer context that needs to be walked in the outer context chain in order to find the context that holds the temporary variable to push.

Executing this instruction searches first in the outer context chain the context to access (it looks for the nth outer context, n being precised in the instruction). Then it pushes on stack the temporary value at the index precised in the instruction from the nth outer context to the top of the active context's stack.

- **StoreRemoteTemp**

Basically this instruction is the same as the PushRemoteTemp instruction with a store instead of a push.

7.3 Exercise

In this exercise you will extend your `ContextInterpreter` with a subclass, `ClosureContextInterpreter`.

Open a Pharo 4 image, load the package SmiLang-ClosureExtension you can find on http://smalltalkhub.com/mc/ClementBera/3DaysVM/main.

Go to the package tag `SmiLang-ClosureExtension>>context` interpreter. A `SmiClosureContextInterpreter` skeleton is given, but many methods are not implemented (they only hold `self shouldBeImplemented`).

The SmiClosureContextInterpreterTests have to pass to proceed to the next chapter.

Good luck and have fun !

CHAPTER 8

Simplified Closures in the Stack Interpreter

In this chapter and the following one, you will introduce the closures in the stack interpreter. For the implementation, we consider as in the chapter 5 that the stack is infinite. It is possible to implement closures on top of a stack using stack pages, but it introduces extra complexity in the code, espscially for non local returns, whereas it does not introduce new concepts.

These two chapters, especially the next one, are **considerably** more complex than the other chapters. As explained in the introduction of this part of the book, they are optional to understand the rest of this book.

8.1 Problems

The naive closure model described in the previous chapter is nice to understand closure implementation but is completely broken in the stack interpreter. The main issue lies with terminated function activations.

A closure has a reference to a function and its enclosing environment. In the case of the context interpreter, the enclosing environment is the outer context. It can happen that a closure outer context is terminated, in which case the closure's activation cannot perform a non local return. However, it can still access the terminated outer context temporaries.

In the stack interpreter no contexts are present. The closure still needs to reference its enclosing environment for remote temporary access and non local return. A naive implementation would be to have the closure referencing directly its outer stack frame. The problem in this case is that when the outer stack frame execution is terminated, the memory location for the stack frame can be overridden by new activations and it is very difficult to know if it has been overridden or not. It is therefore impossible to know if the reference to the outer stack frame still refers to the stack frame or to other data from the stack.

If the outer stack frame happens to be terminated, we have two issues.

1. **Remote temporary access**: Firstly, if the closure's activation tries to access remote temporary variables, the dead outer stack frame may not hold any more the temporary values. In fact, the values may have been overriden by new stack frames.
2. **Non local returns**: Secondly, if the closure's activation tries to perform a non local return, it has to figure out if the closure home context is dead to raise a BlockCannotReturn exception or not. The problem is that the home's stack frame may be dead but also overridden with another stack frame. It is impossible with a direct pointer to know if the data at the memory location of the home stack frame corresponds to the home stack frame or something else.

8.2 Solutions

We identified two problems so we need to describe two solutions. In this chapter, you are going to build a restrictive interpreter that cannot handle non local returns. This implementation solves the remote temporary access problem, but not the one for non local return. In the next chapter, you will enhance the interpreter to support non local returns. The Closure representation you will use for the two stack interpreters with closures have a field named `outerStackFrameWrapper`. You will completely ignore this field in this version of the interpreter (but use it in the next chapter).

This chapter now deals only with remote temporary variables access.

Remote temporary access implementation

If a closure outlives its outer stack frame, the outer stack frame field may have been overridden by new values and we can't access its temporary variables any more. To solve this problem, the compiler will identify temporary variables that are accessed from multiple stack frames (at least one remote access). Such variables values will not be stored on stack, but on an array allocated outside of the stack, also called temp vector, as shown on figure 8.1. In this implementation, the temp vector will be a Pharo array.

When a closure is created, if it needs to access remote temporaries, it keeps a reference to a temp vector in addition to its anonymous function and its outer environment. Specific instructions are added in the language to create temp vectors, to create closures with a temp vector and to support access to

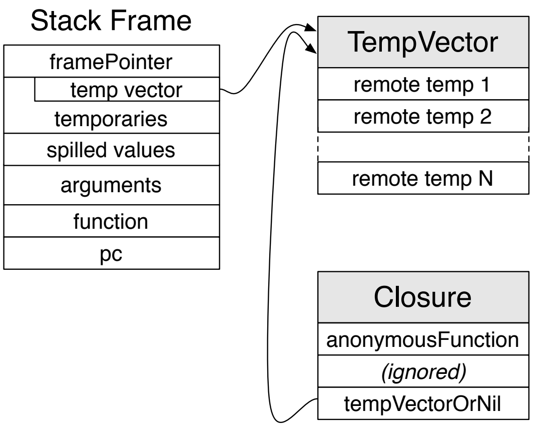

Figure 8.1 Temp vector referenced by a closure and a stack frame.

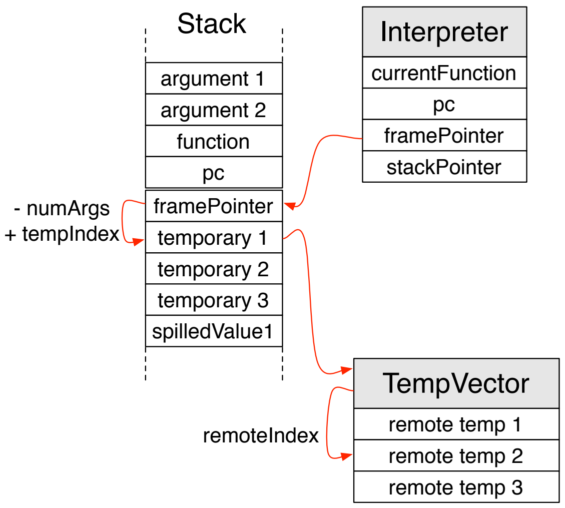

Figure 8.2 Access of the temporary 2 in vector at 3.

temporaries in temp vectors. These instructions are discussed in the next subsection.

If a regular function (non closure function) holds an instruction which creates a closure and that the closure access the regular function temporaries, then the regular function starts with a create temp vector instruction. This instruction creates the temp vector and pushes it on top of stack. The function then stores the temp vector in a temporary variable slot. Further access in the method to the temporaries accessed from multiple stack frame are done indirectly through the temp vector.

Upon closure activation, if the closure has a temp vector reference, the temp vector is pushed on stack after the arguments and before the temporary variables. The instruction accessing remote temporary variables will allow the execution to access the temp vector's values. The instruction holds the index of the tempVector so it can be accessed as a temporary variable and the index of the temporary variable in the temp vector, as show on figure 8.2.

Remote temporary variable example

Let's take an example with a remote temporary access.

Source code:

```
example04ClosureRemoteTemp
  | remoteTemp |
  remoteTemp := 1.
  ^ [ remoteTemp ] value
```

Instructions:

```
example04ClosureRemoteTemp
  <1> createTempVector of size 1
  <2> storeTemp: 1
  <3> pop
  <4> push 1
  <5> storeTemp 1 in vector at 1
  <6> pop
  <7> createClosureWithVectorAt: 1
  <8> activateClosure
  <9> returnTop
```

Closure Instructions:

```
noname
  <1> pushTemp 1 in vector at 1
  <2> returnTop
```

`remoteTemp` is accessed both from the function activation and the closure activation, hence it needs to be put in a temp vector.

1. **Function Instructions 1 to 3**: the interpreter creates a temp vector of size 1 to be able to hold the value of `remoteTemp`. The temp vector is available in the stack frame at the position of the first temporary variable.
2. **Function Instructions 4 to 6**: the interpreter assigns the value 1 to `remoteTemp`. As `remoteTemp` value is stored on a temp vector, it cannot just use a store temp instruction but it needs to use a store remote temp instruction. This instruction precise both the position of the

temp vector in the temporary variable slots of the stack frame and the position of the value of `remoteTemp` in the temp vector.

3. **Function Instruction 7**: the interpreter creates a closure. The closure is initialized with the temp vector in position 1 in the temporary variable slots of the stack frame.
4. **Function Instruction 8**: the interpreter activates the closure. The temp vector held by the closure is pushed on stack just before the temporary variables local to the closure activation.
5. **Closure Instructions**: the interpreter access the remote temporary variable value through the temp vector and answers it.
6. **Function Instruction 9**: the interpreter finally returns 1.

8.3 Extended instruction set

To use closures in the stack interpreter, we extend SmiLang (the non closure interpreter version) with 4 new instructions:

- **CreateTempVector**

Executing this instruction creates a temp vector, represented by a Pharo array in the implemented interpreter, and pushes it on stack. As the temp vector holds temporary variables values, it needs to be initialized with 0 (in SmiLang all the temporaries are initialized with 0). The instruction holds the size of the temp vector to create.

- **CreateVectorClosure**

This instruction represents a closure creation. Executing this instruction creates a new closure and pushes the value on top of the stack. The closure is created with the anonymous function of the instruction and the context that created the closure (the active context at closure creation time).

In addition, this instruction requires the interpreter to store a reference to the temp vector in the closure. The temp vector holds the remote temporaries. The temp vector to store can be found in the current stack frame, it is accessed as a temporary variable using the tempVectorIndex precised in this instruction. If no tempVectorIndex is specified (tempVectorIndex = nil), then the closure does not need any temp vector.

- **PushVectorTemp**

This instruction holds two indexes: the index of the temp vector and the index of the temporary variable in the temp vector. Executing this instruction is done in 2 steps. Firstly, the interpreter fetches the temp vector. It is can be accessed in the stack frame as a temporary variable using as index tempVectorIndex (precised by this instruction). Then it can push on stack the value available in the temp vector at the correct index, precised in tthis instruction (tempIndex).

- **StoreVectorTemp**

Same as pushVectorTemp instruction but mutates the temp vector instead of pushing a value on stack.

8.4 Exercise

In this ultimate exercise you will extend your `StackInterpreter` with a subclass, `SimplifiedClosureStackInterpreter`. You will subclass directly the `StackInterpreter` and not the one with stack pages. It is possible to have a closure stack interpreter with stack pages, but you don't need to deal with the complexity of both to understand the concepts.

Open a Pharo 4 image, load the package `SmiLang-StackClosureExtension` you can find on http://smalltalkhub.com/mc/ClementBera/3DaysVM/main.

Go to the package tag `SmiLang-StackClosureExtension>>stack interpreter`. A `SmiSimplifiedClosureStackInterpreter` skeleton is given, but many methods are not implemented (they only hold `self shouldBeImplemented`).

The `SmiSimplifiedClosureStackInterpreterTests` have to pass to proceed to the next chapter.

Good luck and have fun !

CHAPTER 9

Real Closures in the Stack Interpreter

In this chapter we extend the previous interpreter to support non local returns.

The general idea behind this implementation is that we do not want to make normal returns more complex than how they were in the previous interpreter last chapter. We can slow down a bit non local returns, as they are uncommon, but we can't slow down regular returns, which are very common.

If the implementation requires each regular return to do a check, for example checking if there is a closure referencing the stack frame where the return happens and do something if this is the case, the interpreter is then twice slower than the implementation described in this chapter. This implementation, even though fairly complex, does not add any overhead on normal returns. Only non local returns are slower than they could be, but as they are much less common than normal return, the performance is overall way faster.

9.1 Non local return implementation

This is one of the trickiest part of an efficient Smalltalk VM implementation. Now that remote temporary variables are accessed through a temp vector, the closure needs to keep a reference to its outer stack frame activation **only** for non local returns.

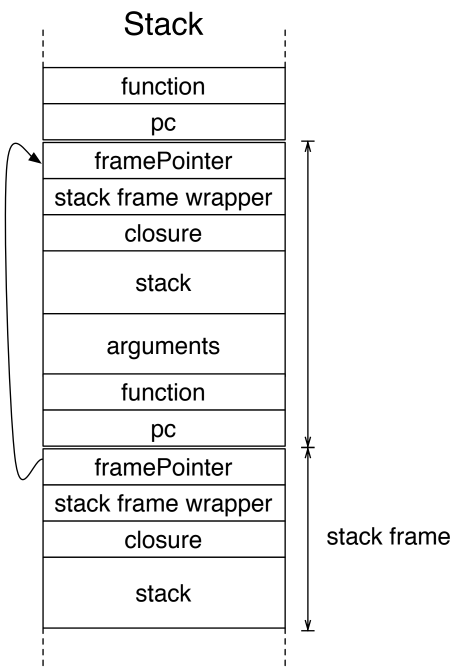

Figure 9.1 New stack frame representation.

Stack frame representation

To support non local returns, we change our stack frame representation. Right after the frame pointer and just before the temporary variable slots, the stack frame keeps two extra fields, as shown on figure 9.1. This implies that accessing temporary variables in this interpreter requires to shift the framePointer by an additionnal 2 for the extra two fields.

Stack frame creation

At stack frame creation time (function call or closure activation), the stack pointer needs to be pushed by an additionnal 2 to make room for the two extra fields. The two fields are used for the potential stack frame wrapper (field 1) and a reference to the closure activated in case of closure activation (field 2).

The two fields are initialized as follow:

1. **Stack frame wrapper**: set to a flag (for example the symbol `#notWrapped`) marking the stack frame as **not** being wrapped as shown on figures 9.2 and 9.3.

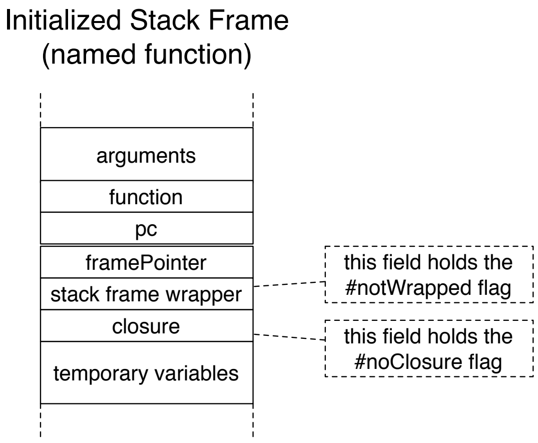

Figure 9.2 Initialized Stack Frame (named function).

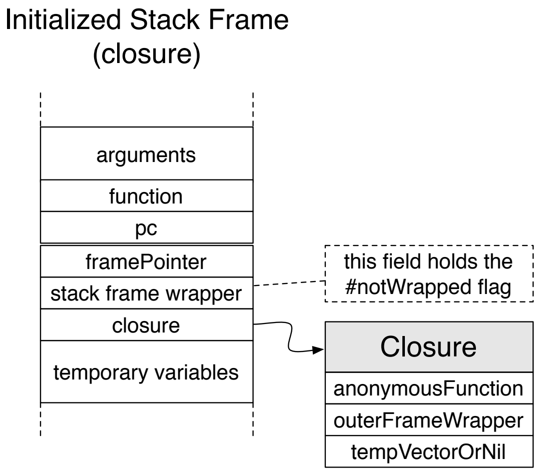

Figure 9.3 Initialized Stack Frame (closure).

2. **Closure**: It depends if the stack frame represent a closure activation or not (we call named function activation a non closure activation):

- *Named function activation*: the closure field is initialized to a flag marking a **non** closure activation, for example `#noClosure`, as shown on figure 9.2.
- *Closure activation*: the closure field is initialized with the closure being activated, as represented on figure 9.3.

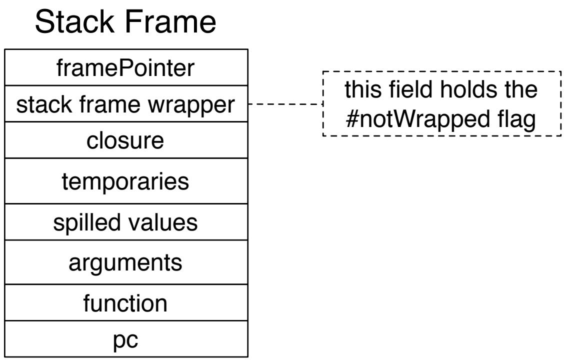

Figure 9.4 Stack Frame (**not** wrapped).

Thanks to the closure field, the interpreter can access the closure of the current activation relatively to the frame pointer or a flag marking a non closure activation (framePointer + 2).

As explained, a stack frame, including a stack frame representing a closure activation, is always initialized **not** wrapped, i.e., with a `#notWrapped` flag in the stack frame pointer field.

Closure creation

When a closure is created, the active stack frame, representing the enclosing environment of the closure, needs to be wrapped (if not already wrapped). Wrapping a stack frame is two step process, a **non** wrapped stack frame is shown on figure 9.4 and a wrapped stack frame is shown on figure 9.5:

1. A **stack frame wrapper** is created. A stack frame wrapper is a data structure allocated aside from the stack. It has a single field that refers to the frame pointer location of the stack frame it wraps. The stack frame wrapper created refers to the active stack frame, representing the enclosing environment of the closure.
2. The stack frame wrapper field of the stack frame is edited to refer to the stack frame wrapper.

The closure, instead of keeping a reference to its outer context as in the context interpreter, keeps a reference to its outer stack frame wrapper as shown on figure 9.5.

Accessing the wrapper from the stack frame

Each stack frame knows if it is already wrapped by checking its stack frame wrapper field:

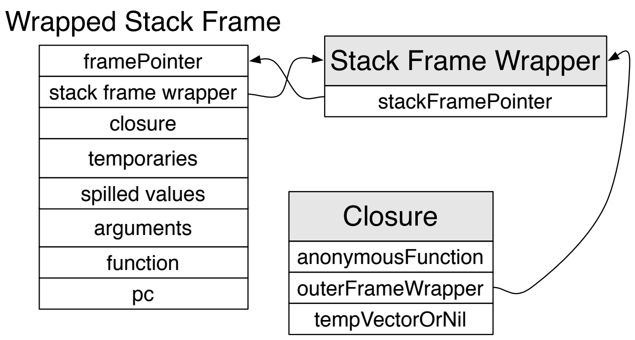

Figure 9.5 Wrapped Stack Frame.

- If the value is the flag set at stack frame creation time, then it is not wrapped, as shown on figure 9.4.
- If the value is not the flag, then it is wrapped and the value of the field is the stack frame wrapper, as described on figure 9.5.

Accessing the stack frame from the wrapper

It's more complex for the stack frame wrapper to access its frame because it needs to know if the stack frame it wraps is still valid. This is a two step operations.

Firstly, the stack frame wrapper needs to figure out if the field on stack it holds is still a frame pointer. If the stack frame has been terminated, it may have been overrided by other frames of different sizes and the field held may be a random field in another stack frame. To do so, the stack frame wrapper walks the stack starting at the interpreter's active frame pointer from a frame pointer to the next one, until it finds a frame pointer matching the field it holds. If it finds a matching frame pointer, then the stack frame wrapper references a stack frame, but the stack frame wrapper does not know yet if it references the stack frame it wraps or another one. If no matching frame pointer is found, then the stack frame is dead.

Secondly, the stack frame wrapper now knows it references a stack frame. It can then read the stack frame wrapper field of the stack frame. If the field refers back to the stack frame wrapper itself, then the correct stack frame is referenced. If it refers to the `#notWrapped` flag or another stack frame wrapper, then the stack frame is dead and has been overridden by another stack frame.

Performing a non local return

As described in the previous subsection, to execute this instruction the interpreter needs to walk up the outer stack frame chain to find the home stack frame. If any outer stack frame relation is broken (a stack frame wrapper does not refer to its stack frame any more), then a BlockCannotReturn exception is raised. Once the home stack frame is found, the execution can return the the home's sender stack frame, pushing the correct non local value on top of its stack.

9.2 Exercise

In this last exercise you will extend your `SimplifiedClosureStackInterpreter` with a subclass, `RealClosureStackInterpreter`.

Open a Pharo 4 image, load the package `SmiLang-StackClosureExtension` you can find on http://smalltalkhub.com/mc/ClementBera/3DaysVM/main.

Go to the package tag `SmiLang-StackClosureExtension>>stack interpreter`. A `SmiRealClosureStackInterpreter` skeleton is given, but many methods are not implemented (they only hold `self shouldBeImplemented`).

The `SmiRealClosureStackInterpreterTests` have to pass to proceed to the next chapter. This exercise is more difficult than the previous ones because of the non local returns implementation.

Good luck and have fun !

9.3 Discussion: non local returns and stack pages

If the interpreter supports both non local returns and stack pages, there are a few more difficulties. The stack frame wrapper, to refer to a field on stack, needs to refer to a stack page and the index of the field in the stack page instead of just the index of the field on stack. In addition, when a stack frame wrapper checks if the field it holds is indeed a frame pointer, the interpreter needs to walk up the stack including stack page switches.

CHAPTER 10

Discussion

10.1 What about Contexts and Stack frames in Smalltalk ?

In Smalltalk, from the language, the programmer has the impression that a context interpreter is running. This is convenient as the programmer can manipulate the contexts as any other objects. This feature is used for the implementation of the debugger, exceptions and continuations.

In the VM, the interpreter is implemented as a stack interpreter for performance. Stack frames are mapped to contexts on demand.

10.2 Context-to-Stack Mapping

The mapping from stack frames to context on demand is usually referred as the **Context-to-Stack Mapping**. Imagine that on all your interpreter, in addition to the public API `interpreter:`, another API called `currentContext` would be available. In the case of the context interpreters, the method currentContext would be implemented as a getter:

```
SmiContextInterpreter >> currentContext
  ^ currentContext
```

However, in the case of the stack interpreter, the interpreter needs to recreate a context from its state and the state of the stack. All the information the context needs is available:

- The temporaries and spilled values are available on the active stack frame.
- The arguments are vailable on the previous stack frame.

- The program counter and the function are available directly in the interpreter variables for the active stack frame, or on stack for the other stack frames.
- The sender can be *lazily* recreated from the previous stack frame.

All the fields can be read that way, and at the exception of the sender field, all the fields can be written into. The only problem remaining is how to set the sender field of a context mapped to a stack frame. This last point is done by splitting the active stack page in two, between the stack frame having its sender set and the previous one, and the sender field can then be set by changing for a given stack page the stack page it has to return to in case of a return across stack pages.

Obviously, as execution of the program keeps going, the state of the context object as to change. To avoid having to edit the context objectc at each instruction executed, the context recreated is usually a proxy to the stack frame. Referencing a stack frame is not possible directly, but it can be done through a technique similar to the stack frame wrapper explained in the preious chapter.

For more information about the Context-to-Stack Mapping, you can read on the Cog blog[1] the articles named **Under Cover Contexts and the Big Frame-Up** and **An Arranged Marriage** that deals precisely with this problem.

10.3 What about Smalltalk Context and sends compared to SmiLang ?

A Smalltalk send does a lookup to find the method to activate and then activate the method. In SmiLang, there is no lookup as the calls are static, but the function activation in SmiLang is very similar to the Smalltalk method activation.

About the contexts, there are one main difference. In Smalltalk, the receiver is for the function activation the equivalent of the first argument. However, the receiver is special as it's the only variable from which the interpreter can directly access its instance variables and it's the variable used for the lookup, so it is stored in a specific field called `receiver`.

For performance, the Pharo implementation of context chose to inline directly the array used by the context's stack in the context object (context are initialized with a fixed number of slots that can be accesed with `at:` and `at:put:`). Closures are slightly more complex in Pharo than in SmiLang, because there are a few optimisations to decrease their memory footprint and improve their performance.

[1] http://www.mirandabanda.org/cogblog/

You can now look into the `Context` class in Pharo and verify that you understand what the instance variables correspond to and how they are used. You can also inspect some contexts and verify that you understand their states.

10.4 Closures in mainstream languages

As closures are becoming more and more popular, mainstream programming languages such as Java adopted them. Many of these languages have closures able to access remote temporary variables but not performing non local returns. This implies that the closure representation can be simplified to an anonymous function and a temp vector, as the outer context / outer stack frame wrapper is not needed any more.

A few languages, like Ruby, allow the programmer to use non local returns in their closures.

10.5 Correction

In the repository http://smalltalkhub.com/mc/ClementBera/3DaysVM/main, one can find a package named SmiLang-Correction. It includes a *possible* implementation for each interpreter.