Creating Context In ZagSmalltalk

Daniel Franklin & Dave Mason
Toronto Metropolitan University

This talk

Our claim is that lazy creation of contexts saves a significant number of instructions in a typical program.

We plan to demonstrate that with a very untypical program!

Fibonacci Example

In Smalltalk

Fibonacci stresses a programming languages handling of multiple calls

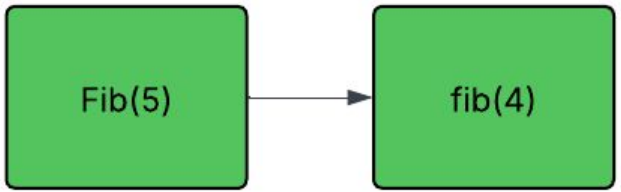
```
fib
    self <= 2 ifTrue: [^1].
    ^ (self - 1) fib + (self - 2) fib
```

Fibonacci Example: 5 fib

```
fib
    self <= 2 ifTrue: [^1].
    ^ (self - 1) fib + (self - 2) fib
```

5 fib

-> 4 fib



Fibonacci Example: 5 fib

```
fib
    self <= 2 ifTrue: [^1].
    ^ (self - 1) fib + (self - 2) fib
```

5 fib

-> 4 fib

-> 3 fib

Fibonacci Example: 5 fib

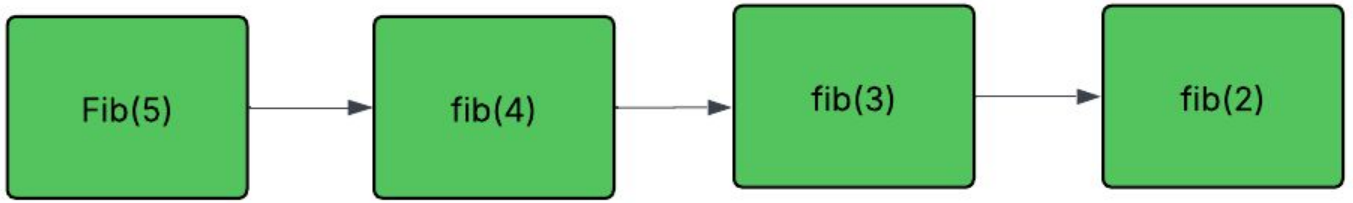
```
fib
    self <= 2 ifTrue: [^1].
    ^ (self - 1) fib + (self - 2) fib
```

5 fib

-> 4 fib

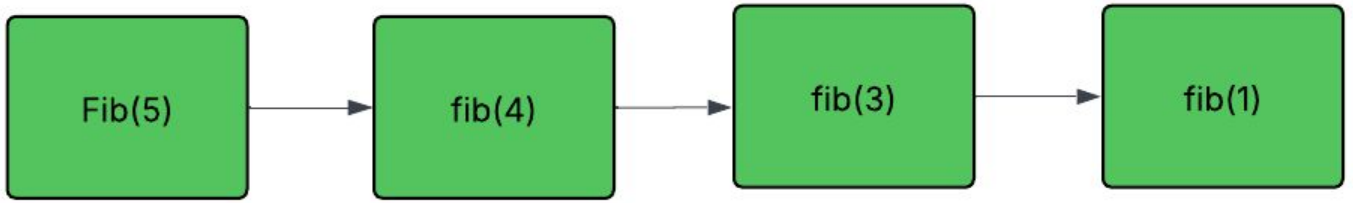
-> 3 fib

-> 2 fib return 1



Fibonacci Example: 5 fib

```
fib
    self <= 2 ifTrue: [^1].
    ^ (self - 1) fib + (self - 2) fib
```



5 fib

-> 4 fib

-> 3 fib

-> 2 fib return 1

-> 1 fib return 1

Fibonacci Example: 5 fib

```
fib
    self <= 2 ifTrue: [^1].
    ^ (self - 1) fib + (self - 2) fib
```

5 fib

-> 4 fib

-> 3 fib return 2

-> 2 fib return 1

-> 1 fib return 1

Fibonacci Example: 5 fib

```
fib
    self <= 2 ifTrue: [^1].
    ^ (self - 1) fib + (self - 2) fib
```

5 fib

-> 4 fib

-> 3 fib return 2

-> 2 fib return 1

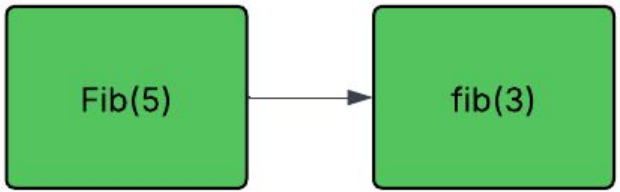
Fibonacci Example: 5 fib

```
fib
    self <= 2 ifTrue: [^1].
    ^ (self - 1) fib + (self - 2) fib
```

5 fib

-> 4 fib return 3

-> 3 fib



Fibonacci Example: 5 fib

```
fib
    self <= 2 ifTrue: [^1].
    ^ (self - 1) fib + (self - 2) fib
```

5 fib

-> 4 fib return 3

-> 3 fib

-> 2 fib return 1

Fibonacci Example: 5 fib

```
fib
    self <= 2 ifTrue: [^1].
    ^ (self - 1) fib + (self - 2) fib
```

5 fib

-> 4 fib return 3

-> 3 fib

-> 2 fib return 1

-> 1 fib return 1

Fibonacci Example: 5 fib

```
fib
    self <= 2 ifTrue: [^1].
    ^ (self - 1) fib + (self - 2) fib
```

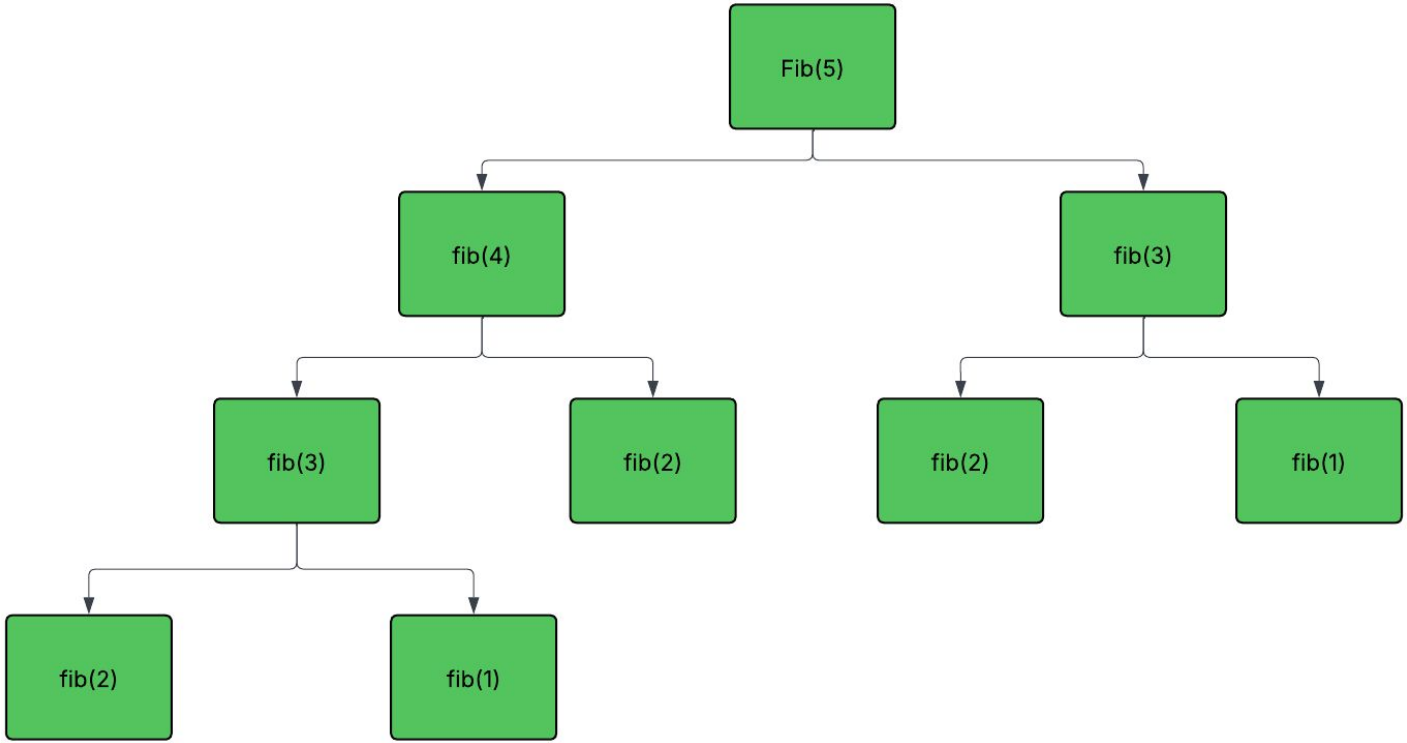
5 fib

-> 4 fib return 3

-> 3 fib return 2

and finally 3+2 results in 5 the solution

Example: 5 fib



What is a context?

A context contains the information needed to execute a method

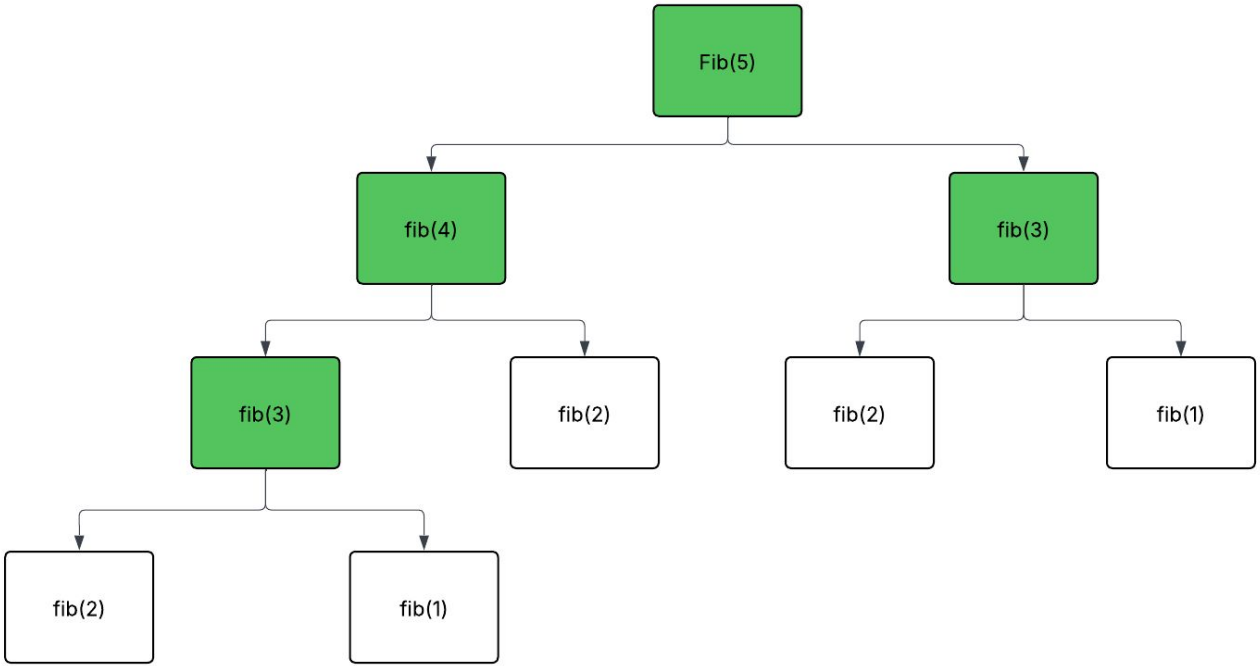
- self, parameters and any space for locals
- reference to the return location in the caller to resume execution
- a pointer to the method

Contexts are not free.

Most contexts are ephemeral - the method will return after 1 or two message sends - sometimes 0 - so the less code we can set up, the better

Zag Smalltalk's approach

Making a full context only on demand eliminates 50% of the contexts created here.



Phases of contexts

1. Initially a method call includes a reference to our caller's context and enough information to build a new context if needed.
2. If a send or store-to-local operation is executed in the method we push a context onto the stack; it points to the caller's context and the method
    - being on the stack, several fields of the context remain uninitialized
    - costs ~15 instructions
3. if the context gets spilled to the heap, the context must become a complete object
    - The contexts and context data are not contiguous so context must have a pointer to context data
    - costs ~15 more instructions to complete the object and on the order of 100 to allocate and copy it to the heap

Some background on the memory model in Zag Smalltalk

The memory model includes

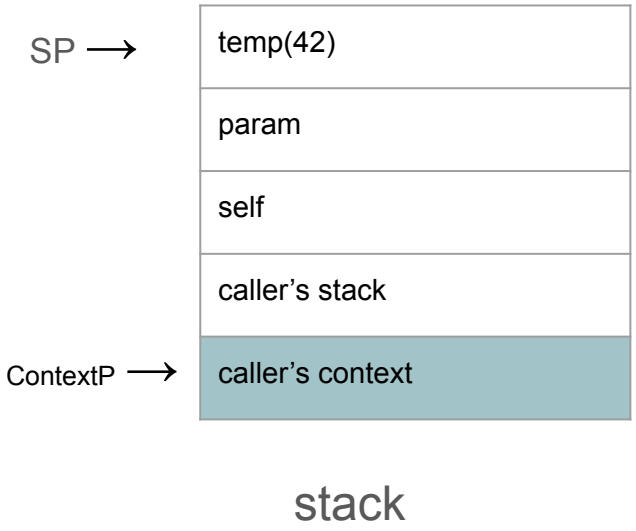
- a stack of finite size (~1000 words)
- a heap for complex objects
- when the stack space is exhausted the stack is spilled/copied to the heap

The stack is linear and contiguous as opposed to a linked list model.

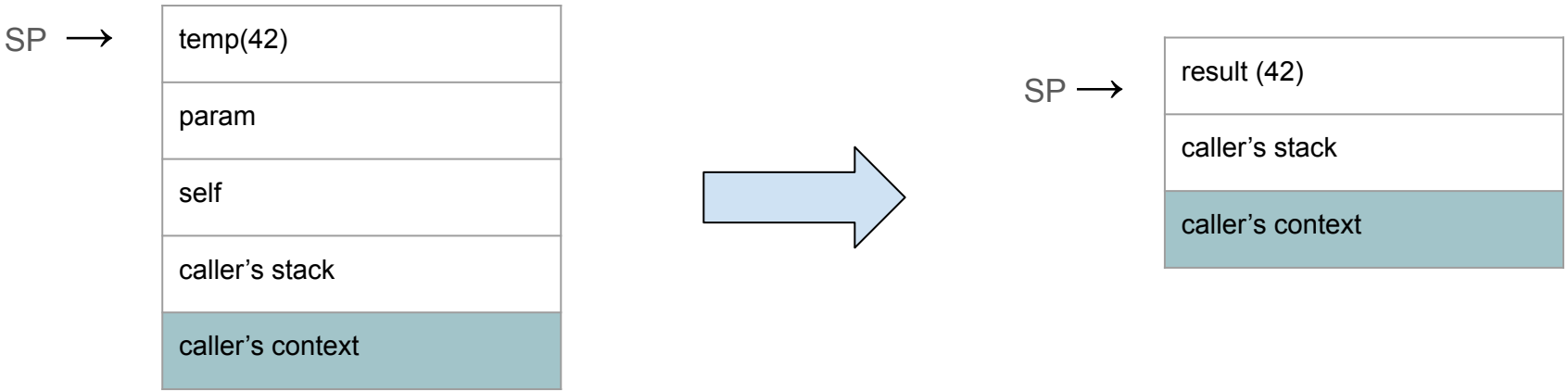
Stack objects only point outfard (to caller contexts or to heap objects), heap objects never point to stack objects.

Stack state in a method, Phase 1

```
converter: param

    ^ 42
```

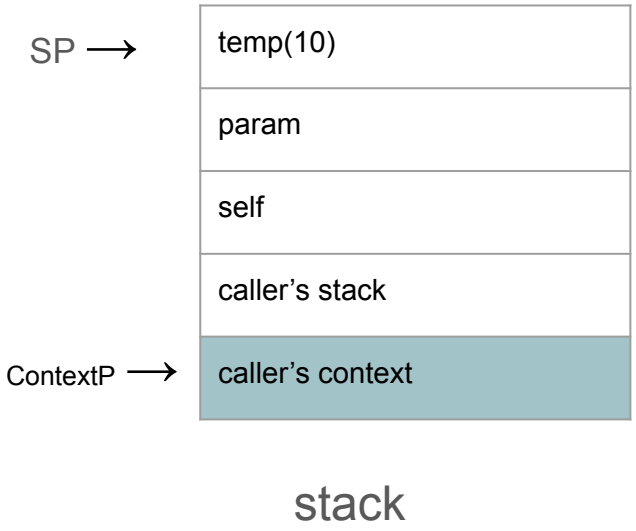


Stack when the call returns from Phase 1

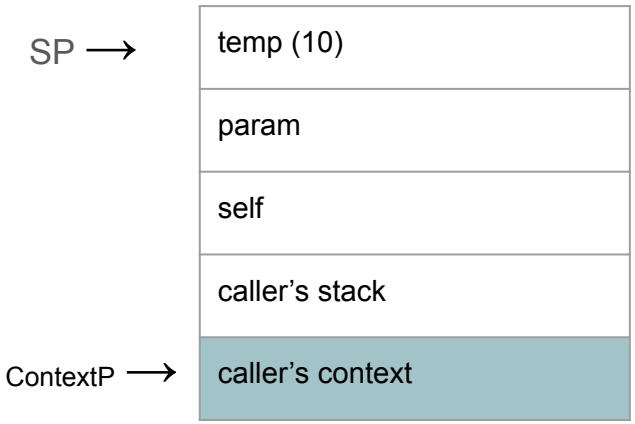


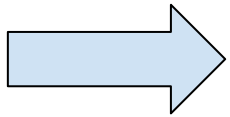
Stack state in a method

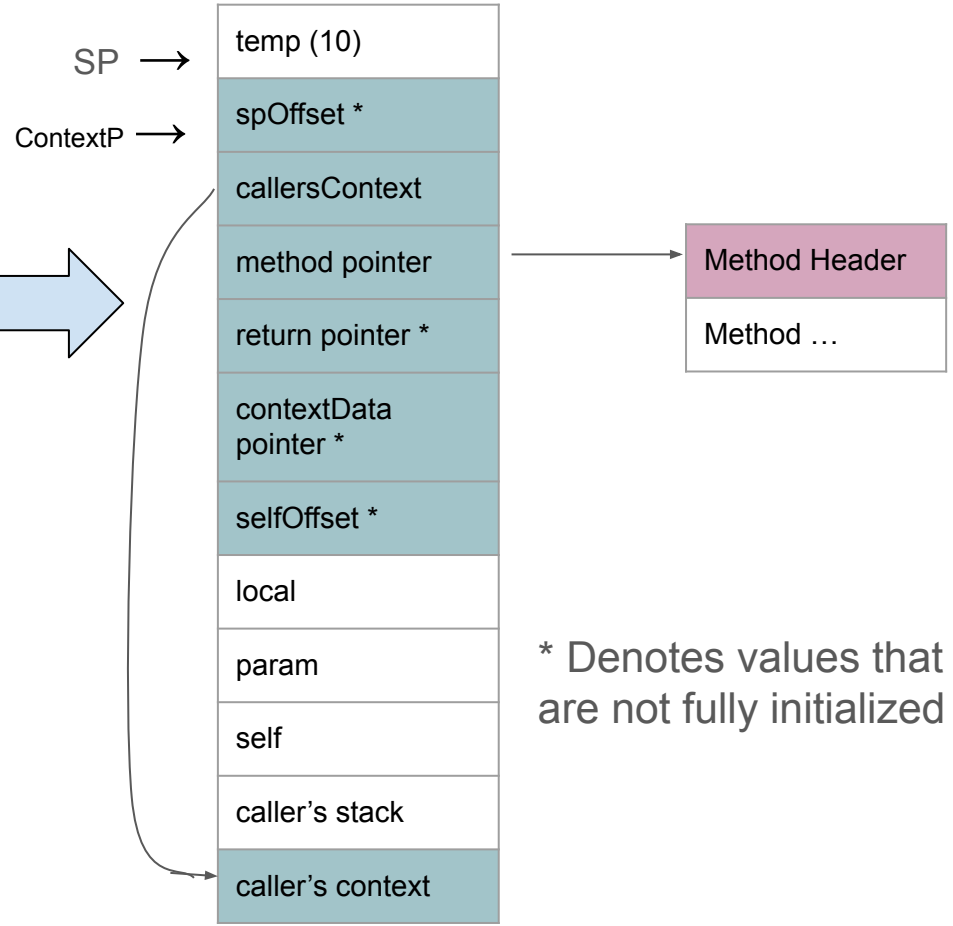
```
converter: param

        | local |

        local := 10.

        ^ local negated
```



Stack state before and after pushing a context, Phase 2



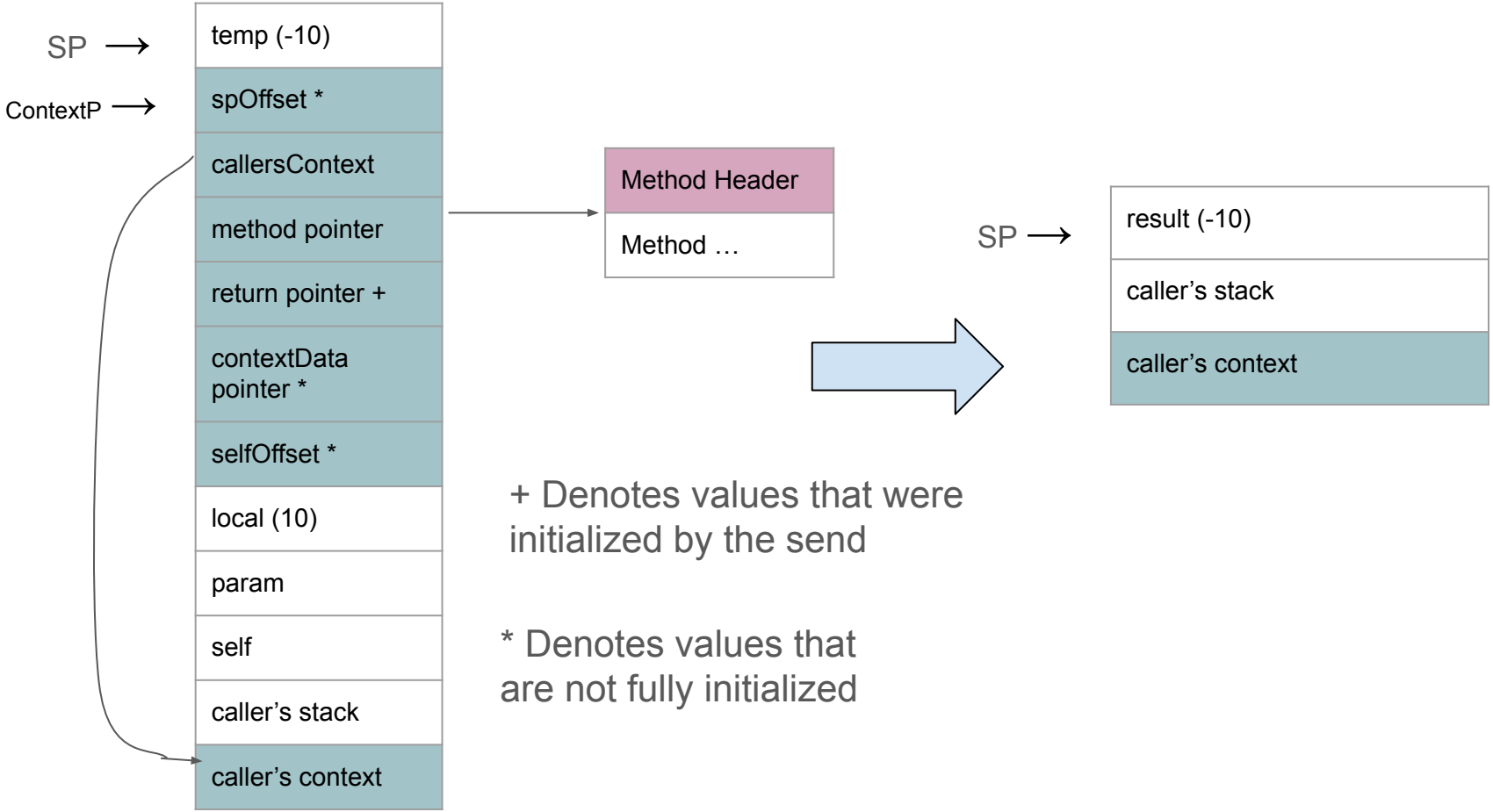




Approximately 15 instructions

* Denotes values that are not fully initialized

Stack when the call returns - from Phase 2



+ Denotes values that were initialized by the send

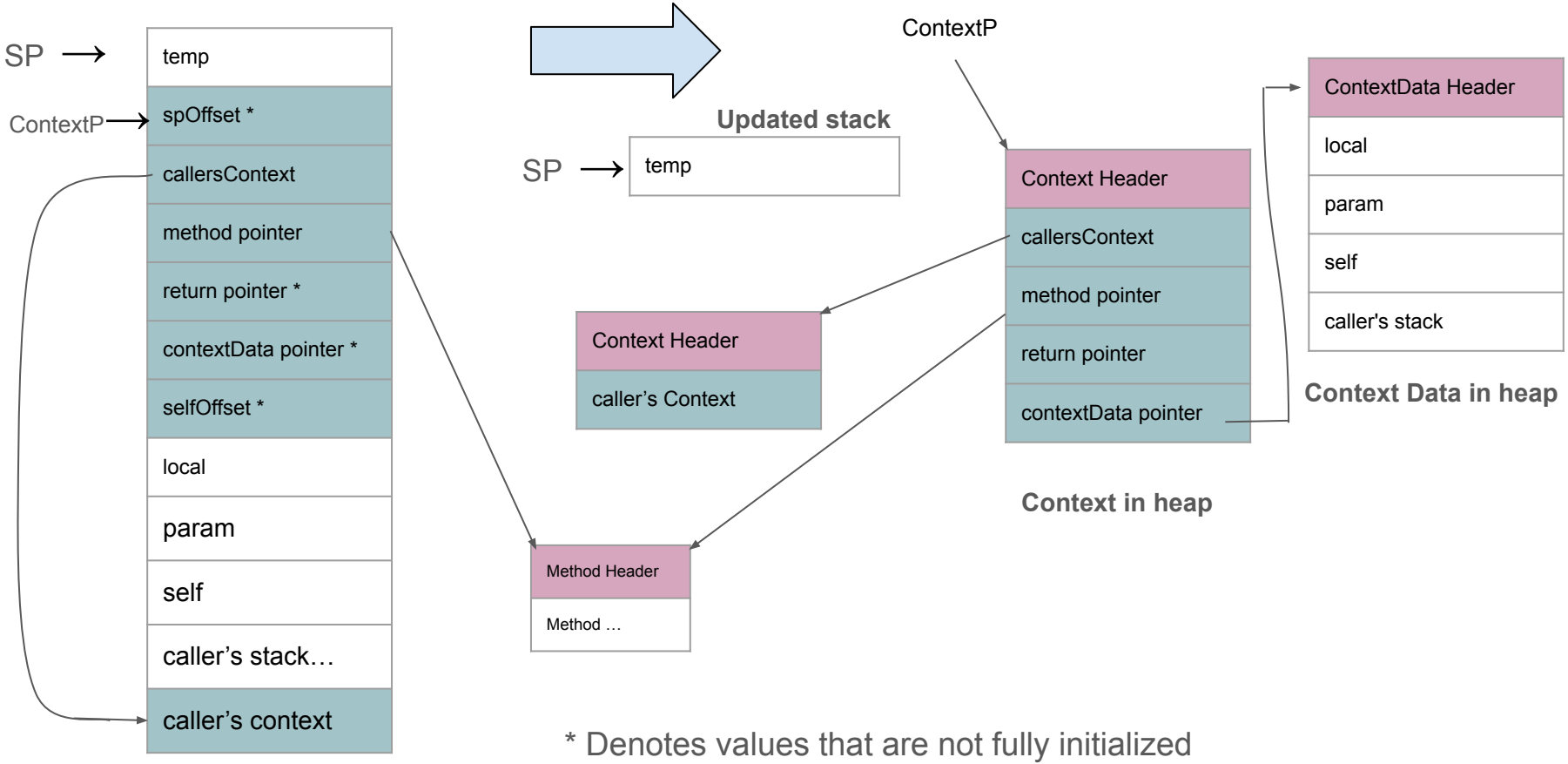
* Denotes values that are not fully initialized

Spilling

Stacks are finite in space

- During execution every stack push operation needs to test if the stack has enough space to store the object and if not the stack must be spilled to the heap.

Stack state before and after spilling to heap, Phase 3



* Denotes values that are not fully initialized

So far so good

The message sends we have been looking at have been fairly simple.

One feature of Smalltalk is the frequent use of closures also know as blocks.

Blocks introduce a complexity when creating contexts because Blocks are able

- to modify the state of the defining context;
- perform non-local returns from arbitrary locations back to the method that defined them

Smalltalk Blocks

The fib example passed the block `[^1]` to the `ifTrue:` method which

- If the target of the `ifTrue:` is `true`, evaluate the block by sending the `value` message to the block
- Otherwise the target of the `ifTrue:` is `false` which returns without evaluating the block

Blocks may be passed around and executed independently of the calling method.

Blocks also have access to all of the defining method's values via the context data.

Smalltalk Blocks

In the fib example **`self <= 2`** returns a boolean value **`true`**/**`false`**,

The boolean is then sent the message **`ifTrue:`** with the argument `[^1]`.

> **^** is the symbol for return

The True implementation of the **`ifTrue:`** method will evaluate the block by sending it a **`value`** message.

The False implementation does not evaluate the block and simply returns.

Non-Local Returns

Smalltalk can pass a block as an argument and the block can contain a `^` return

- The block `[^1]` has a nonlocal return and must reference the defining method's context because this return will result in a return to the defining method's caller no matter where the block is evaluated.
- Blocks without non-local returns will not reference the defining method's context and instead will return to the caller which sent it the **`value`** message.

Non-Local Returns

Common Lisp, Scheme, Ruby and Kotlin also support non-local returns

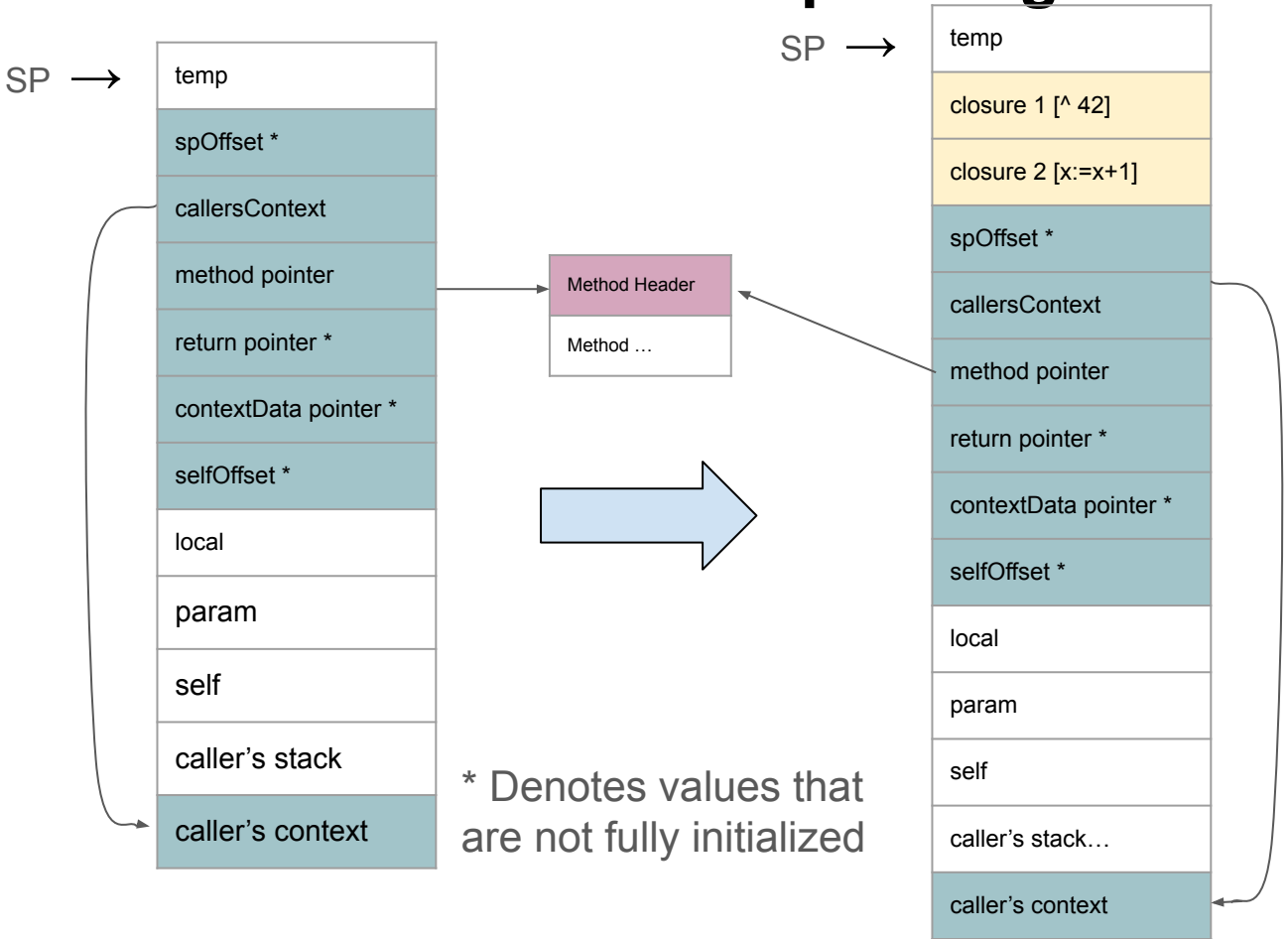
C and C++ support setjmp/longjmp to exit multiple stack frames

Java, Python, C#, JavaScript have exceptions that mimic non-local return behaviour where the execution will jump to the first catch handler found for the type of exception

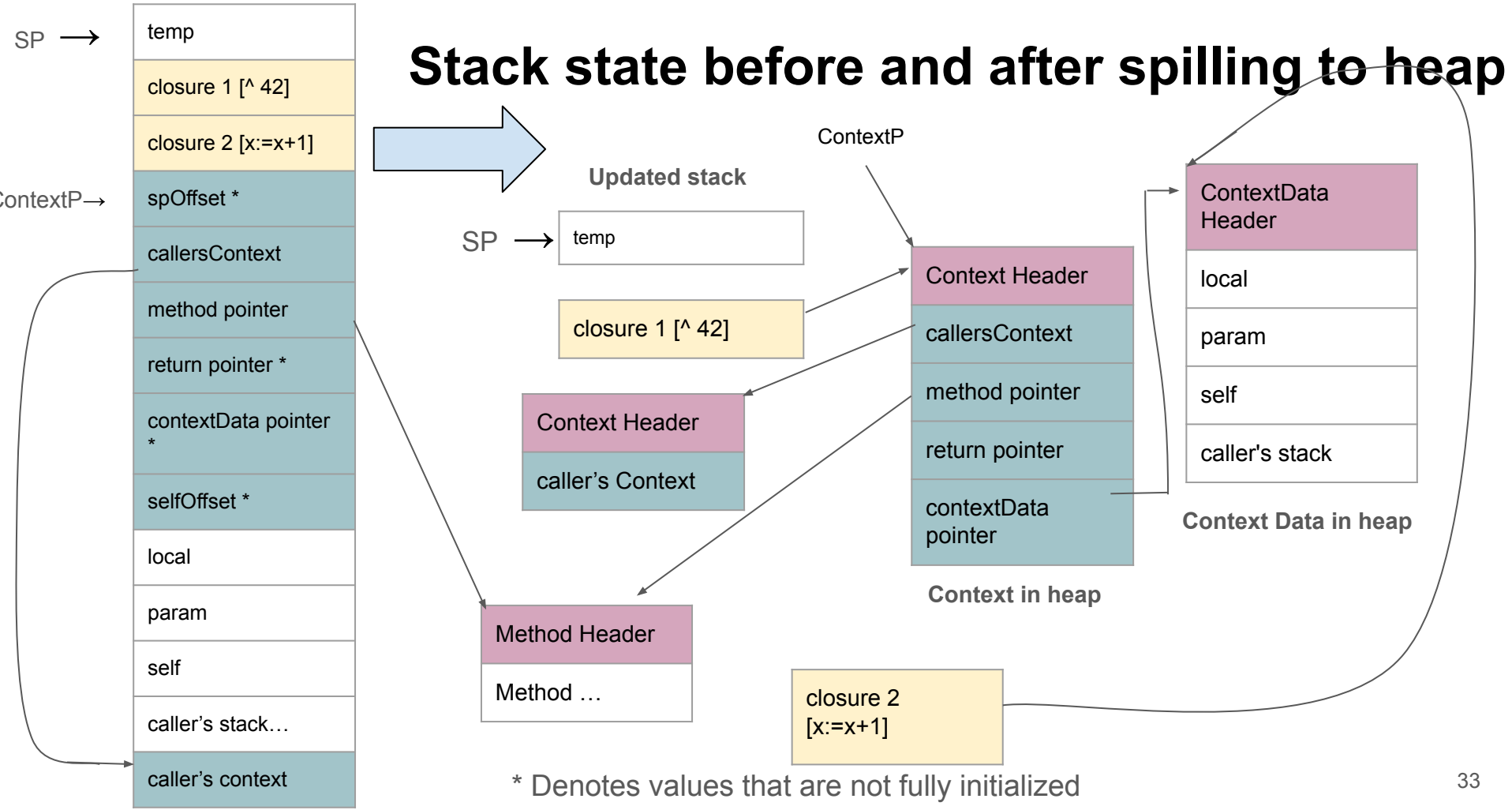
Blocks are able to read and modify data from caller

```
converter: aNumber


    | x |
     x := aNumber * 3.
    "Block reads and writes to x
    25 > x ifTrue: [ ^ 42 ] ifFalse: [ x := x + 1 ].
    ^ x
```

Stack state before and after pushing closures



Stack state before and after spilling to heap



* Denotes values that are not fully initialized

What is Zag Smalltalk?

Smalltalk is a 100% object oriented dynamic programming language.

we are writing a new compiler and runtime for Smalltalk called Zag Smalltalk, https://github.com/Zag-Research/Zag-Smalltalk

Smalltalk has a syntax that can fit on a card, but supports complex language features like non-local returns

Conclusions

Our claim was that lazy creation of contexts saves a significant number of instructions in a typical program.

Our multi-phase implementation achieves that goal.

Questions

Daniel Franklin <daniel.franklin@torontomu.ca>

Dave Mason <dmason@torontomu.ca>

Zag Smalltalk https://github.com/Zag-Research/Zag-Smalltalk