Interpreters and virtual machines

Advanced Compiler Construction

Michel Schinz – 2020–04–23

Interpreters

Interpreters

An **interpreter** is a program that executes another program, which could be represented as:

- raw text (source code), or
- a tree (AST of the program), or
- a linear sequence of instructions.

Pros of interpreters:

- no need to compile to native code,
- simplify the implementation of programming languages,
- often fast enough on modern CPUs.

Text-based interpreters

Text-based interpreters directly interpret the textual source of the program. Seldom used, except for trivial languages where every expression is evaluated at most once (no loops/functions).
Plausible example: a calculator, evaluating arithmetic expressions while parsing them.

Tree-based interpreters

Tree-based interpreters walk over the abstract syntax tree of the program to interpret it.

Better than string-based interpreters since parsing and analysis is done only once.

Plausible example: a graphing program, which repeatedly evaluates a function supplied by the user to plot it.

(Also, all the interpreters included in the L_3 compiler are tree-based.)

Virtual machines

Virtual machines

Virtual machines resemble real processors, but are implemented in software. They take as input a sequence of instructions, and often also abstract the system by:

- managing memory,
- managing threads,
- managing I/O,
- etc.

Used in the implementation of many important languages, e.g. SmallTalk, Lisp, Forth, Pascal, Java, C#, etc.

Why virtual machines?

Since the compiler has to generate code for some machine, why prefer a virtual over a real one?

- for portability: compiled VM code can be run on many actual machines,
- for simplicity: a VM is usually more high-level than a real machine, which simplifies the task of the compiler,
- for simplicity (2): a VM is easier to monitor and profile, which eases debugging.

Virtual machines drawbacks

Virtual machines have one drawback: performance.

Why?

- interpretation overhead (fetching/decoding, etc.).

Mitigations:

- compile the (hot parts) of the program being interpreted,
- adapt optimization on program behavior.

Kinds of virtual machines

Two broad kinds of virtual machines:

– **stack-based VMs** use a stack to store intermediate results, variables, etc.

– **register-based VMs** use a limited set of registers for that, like a real CPU.

What's best?

– for compiler writers: stack-based is easier (no register allocation),

– for performance: register-based *can* be better.

Most widely-used virtual machines today are stack-based (e.g. the JVM, .NET's CLR, etc.) but a few recent ones are register-based (e.g. Lua 5.0).

Virtual machine input

Virtual machines take as input a program expressed as a sequence of instructions:

- each instruction is identified by its **opcode** (**op**eration **code**), a simple number,
- when opcodes are one byte, they are often called **byte codes**,
- additional arguments (e.g. target of jump) appear after the opcode in the stream.

VM implementation

Virtual machines are implemented in much the same way as a real processor:

1. the next instruction to execute is fetched from memory and decoded,
2. the operands are fetched, the result computed, and the state updated,
3. the process is repeated.

VM implementation

Which language are used to implement VMs?

Today, often C or C++ as these languages are:

- fast,
- at the right abstraction level,
- relatively portable.

Moreover, GCC and clang have an extension that can be used to speed-up interpreters.

Implementing a VM in C

```
typedef enum {
  add, /* … */
} instruction_t;

void interpret() {
  static instruction_t program[] = { add /* … */ };
  instruction_t* pc = program;
  int* sp = …; /* stack pointer */
  for (;;) {
    switch (*pc++) {
    case add:
      sp[1] += sp[0];
      sp++;
      break;
      /* … other instructions */
    }
  }
}
```

Optimizing VMs

The basic, `switch`-based implementation of a virtual machine just presented can be made faster using several techniques:

- threaded code,
- top of stack caching,
- super-instructions,
- JIT compilation.

Threaded code

Threaded code

In a `switch`-based interpreter, two jumps per instruction:

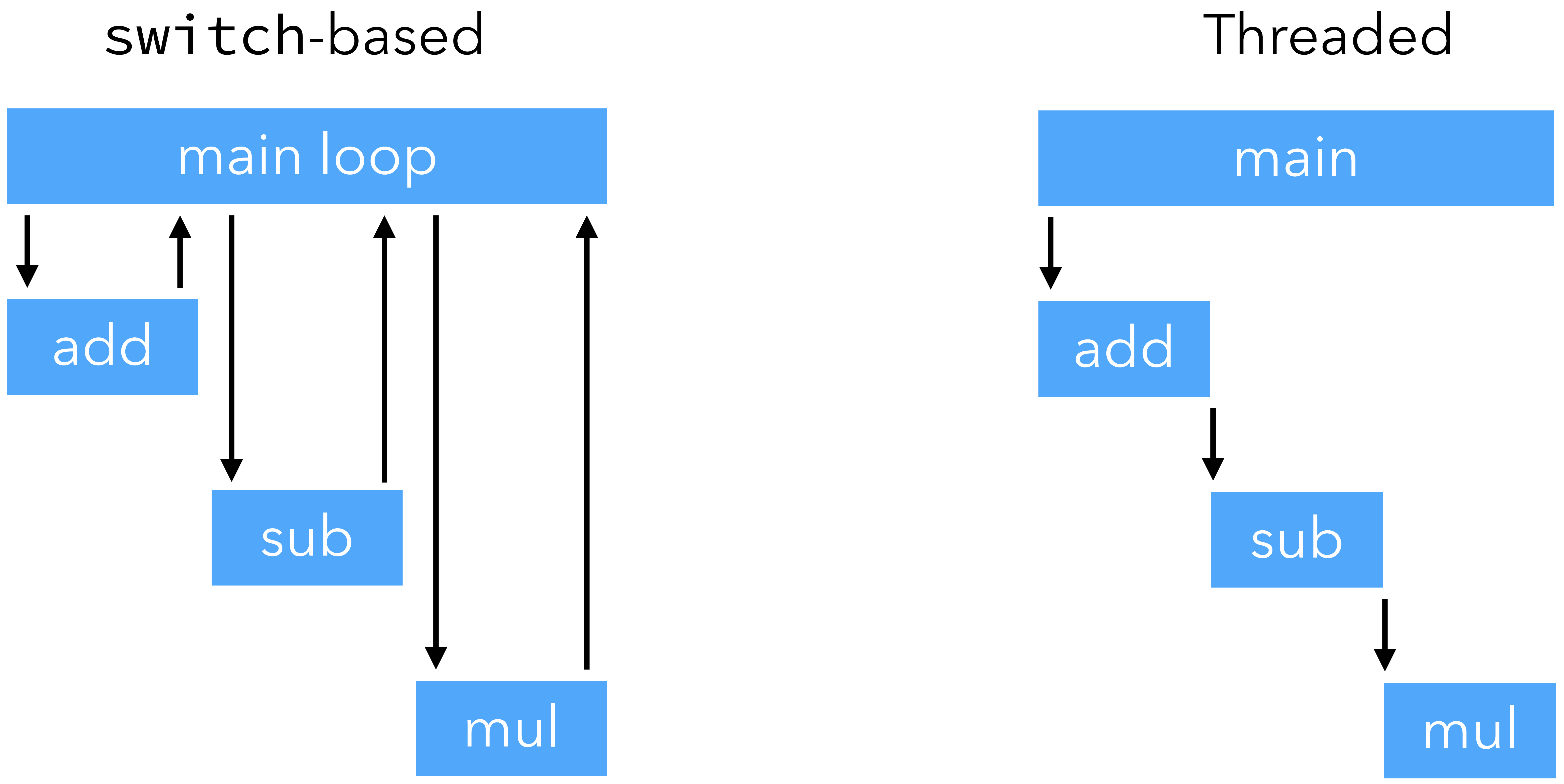
- one to the branch handling the current instruction,
- one from there back to the main loop.

The second one should be avoided, by jumping directly to the code handling the next instruction.

This is the idea of **threaded code**.

Switch *vs* threaded

Program: `add sub mul`



Implementing threaded code

Two main variants of threading:

1. **indirect threading**, where instructions index an array containing pointers to the code handling them,
2. **direct threading**, where instructions are pointers to the code handling them.

Pros and cons:

- direct threading has one less indirection,
- direct threading is expensive on 64 bits architectures (one opcode = 64 bits).

Threaded code in C

Threaded code represents instructions using code pointers.

How can this be done in C?

- in standard (ANSI) C, with function pointers (slow),
- with GCC or clang, with label pointers (fast).

Direct threading in ANSI C

Direct threading in ANSI C:

- one function per VM instruction,
- the program is a sequence of function pointers,
- each function ends with code to handle the next instruction.

Easy but very slow!

Direct threading in ANSI C

```
typedef void (*instruction_t)();
static instruction_t* pc;
static int* sp = …;

static void add() {
  sp[1] += sp[0];
  ++sp;
  (*++pc)(); /* handle next instruction */
}

/* … other instructions */

static instruction_t program[] = { add, /* … */ };

void interpret() {
  sp = …;
  pc = program;
  (*pc)(); /* handle first instruction */
}
```

Direct threading in ANSI C

Major problems of direct threading in ANSI C:

- slower than switch-based,
- stack overflow in the absence of tail call elimination.

With compilers that do not do TCE, the only option is to use trampolines (or similar), which is even slower!

Conclusion: direct threading in ANSI C is not realistic.

Direct threading with GCC

Direct threading with GCC or clang:

– one *block* per VM instruction,

– the program is a sequence of *block* pointers,

– each function ends with code to handle the next instruction.

This requires a non-standard extension called *labels as values* (basically, label pointers).

Direct threading with GCC

label as value

```
void interpret() {
  void* program[] = { &&l_add, /* … */ };

  int* sp = …;
  void** pc = program;
  goto **pc; /* jump to first instruction */
```

computed goto

```
 l_add:
   sp[1] += sp[0];
   ++sp;
   goto **(++pc); /* jump to next instruction */

 /* … other instructions */
}
```

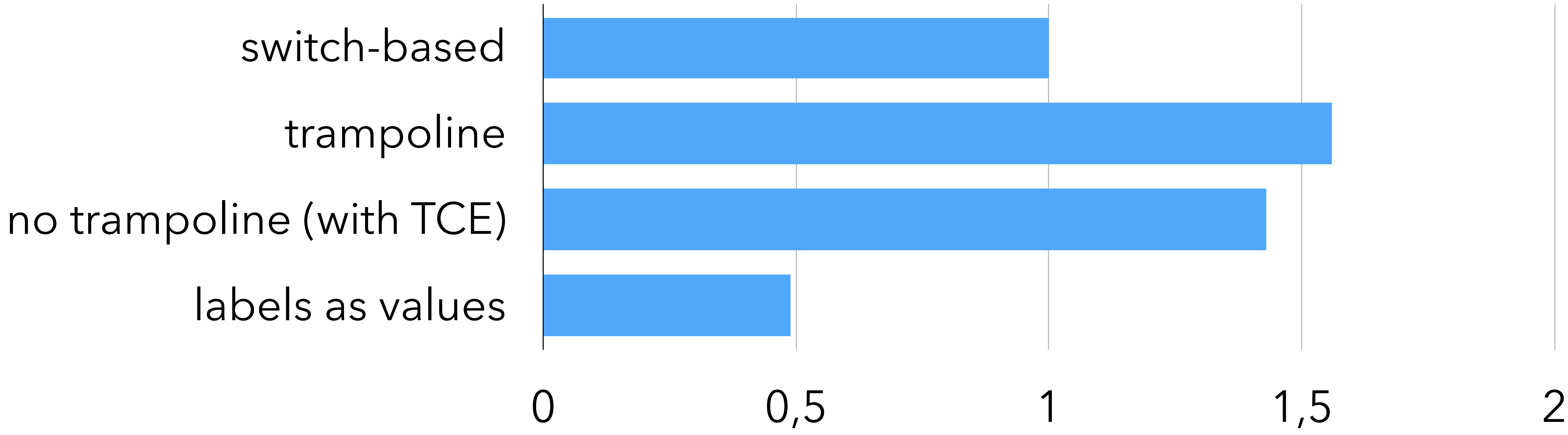
Threading benchmark

Benchmark: 500'000'000 iterations of a loop

Processor: 2.3 GHz Intel Core i9

Compiler: clang 11.0.3

Optimization settings: −O3



Top-of-stack caching

Top-of-stack caching

In a stack-based VM, the stack is typically represented as an array in memory, accessed by almost all instructions.

Idea:

store topmost element(s) in registers.

However:

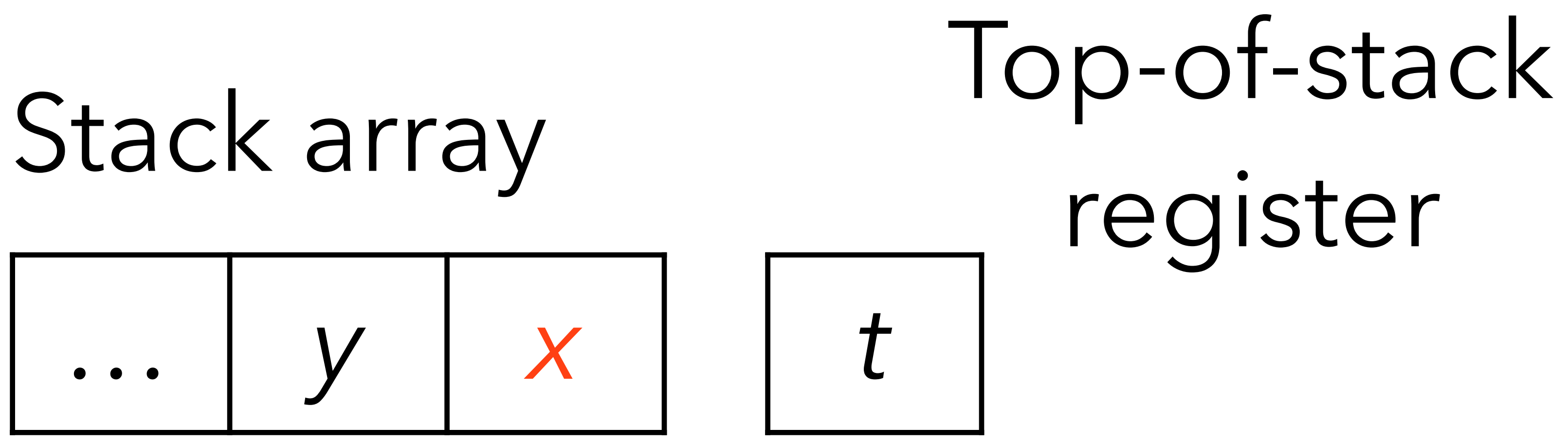
storing a fixed number of topmost elements is not a good idea!

Therefore:

store a variable number of topmost elements, e.g. at most one.

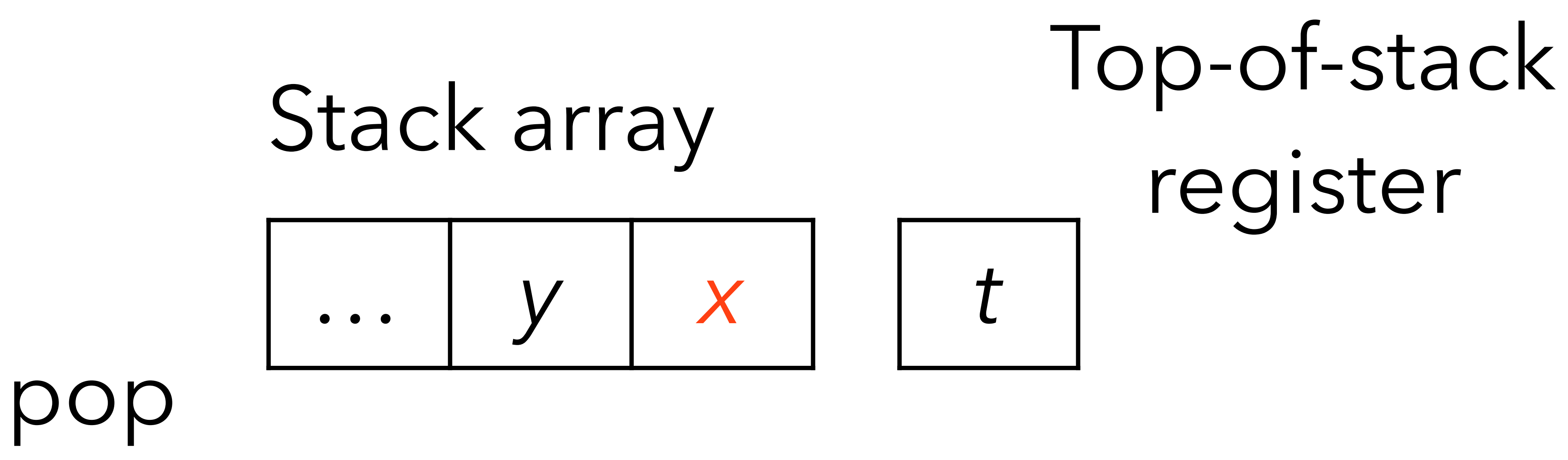
Top-of-stack caching

The top element is always cached:



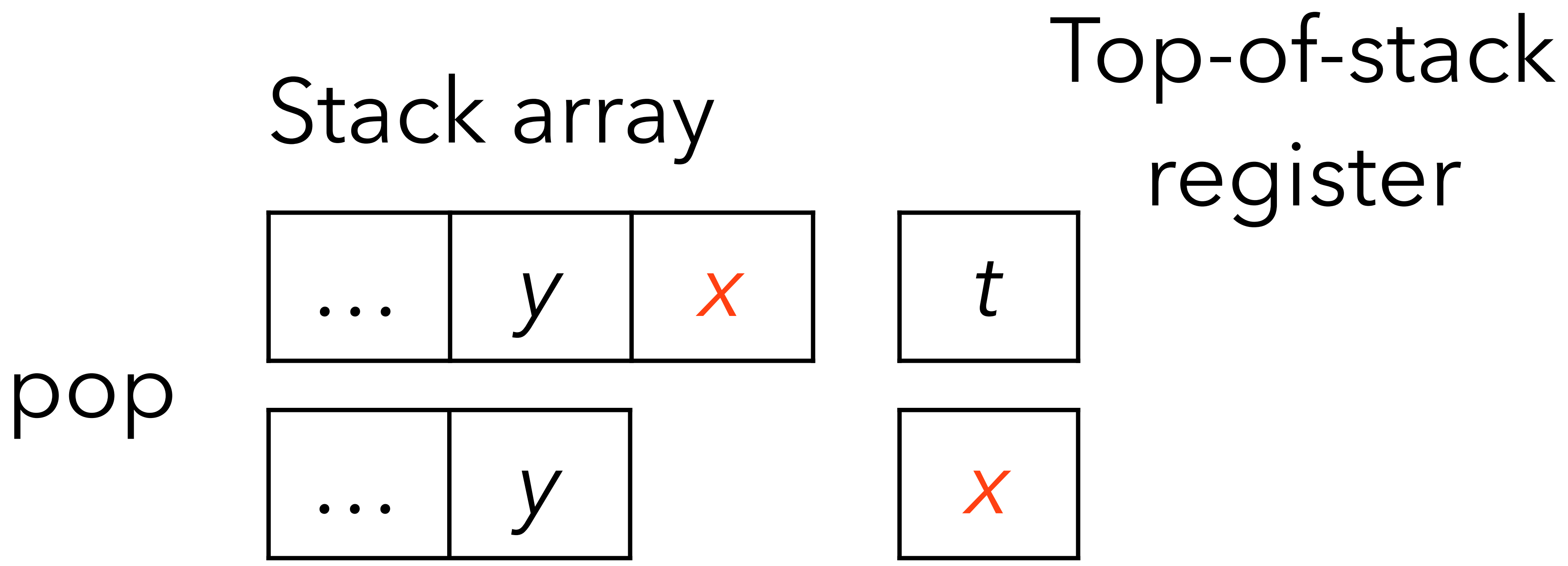
Top-of-stack caching

The top element is always cached:



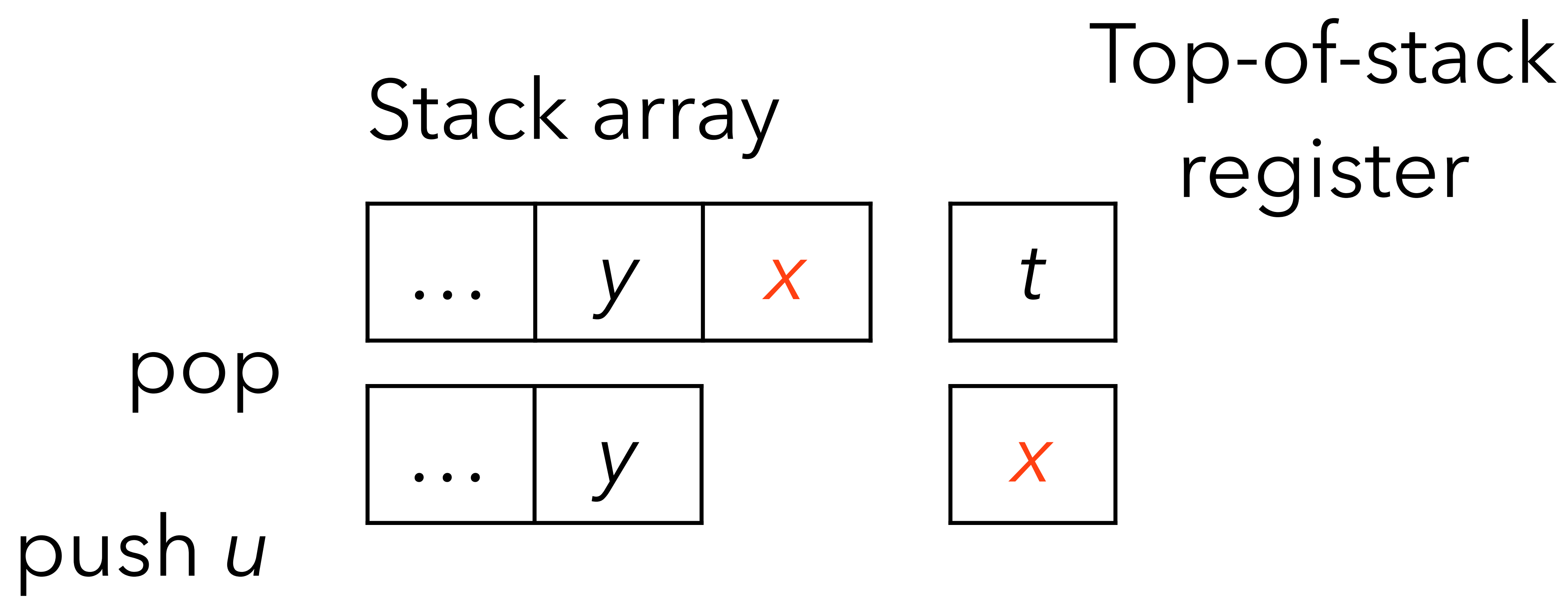
Top-of-stack caching

The top element is always cached:



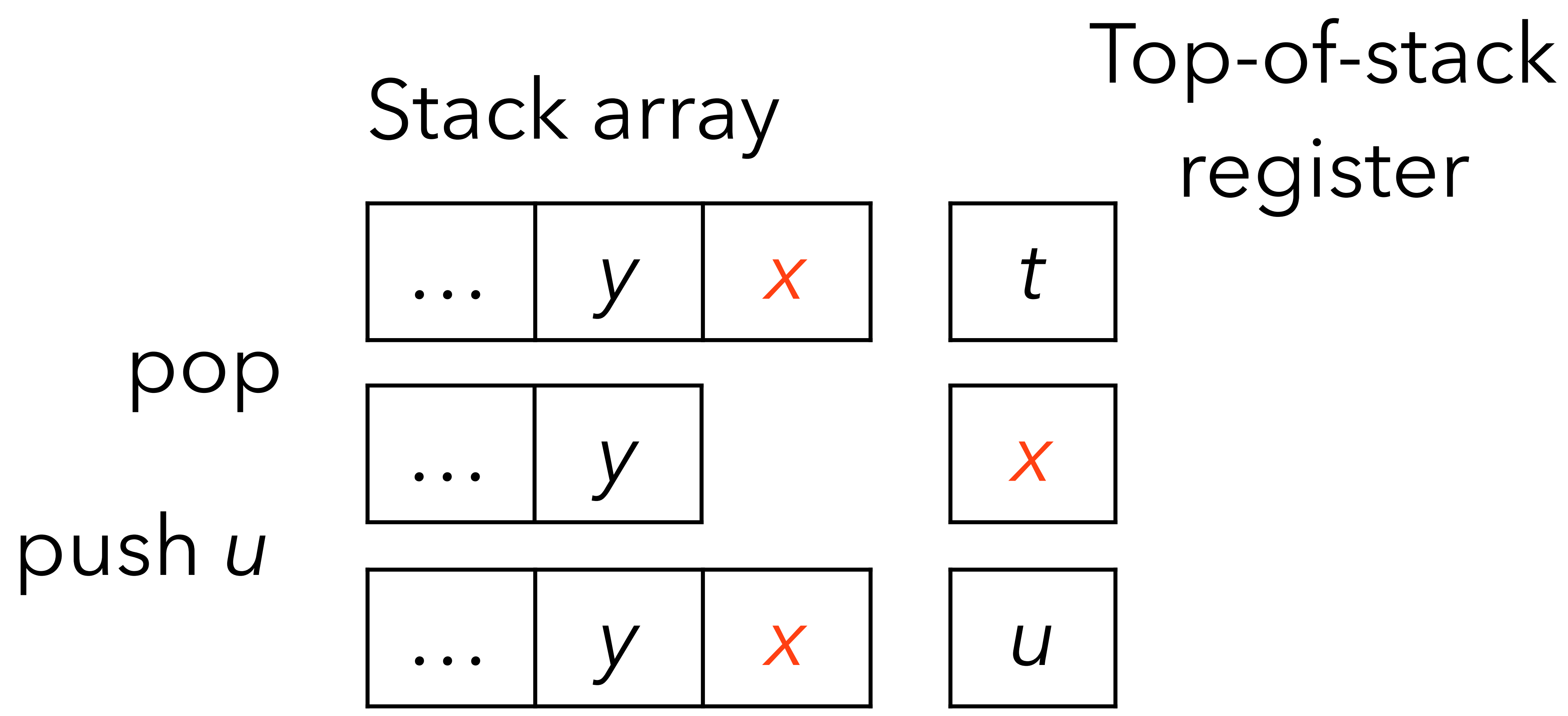
Top-of-stack caching

The top element is always cached:



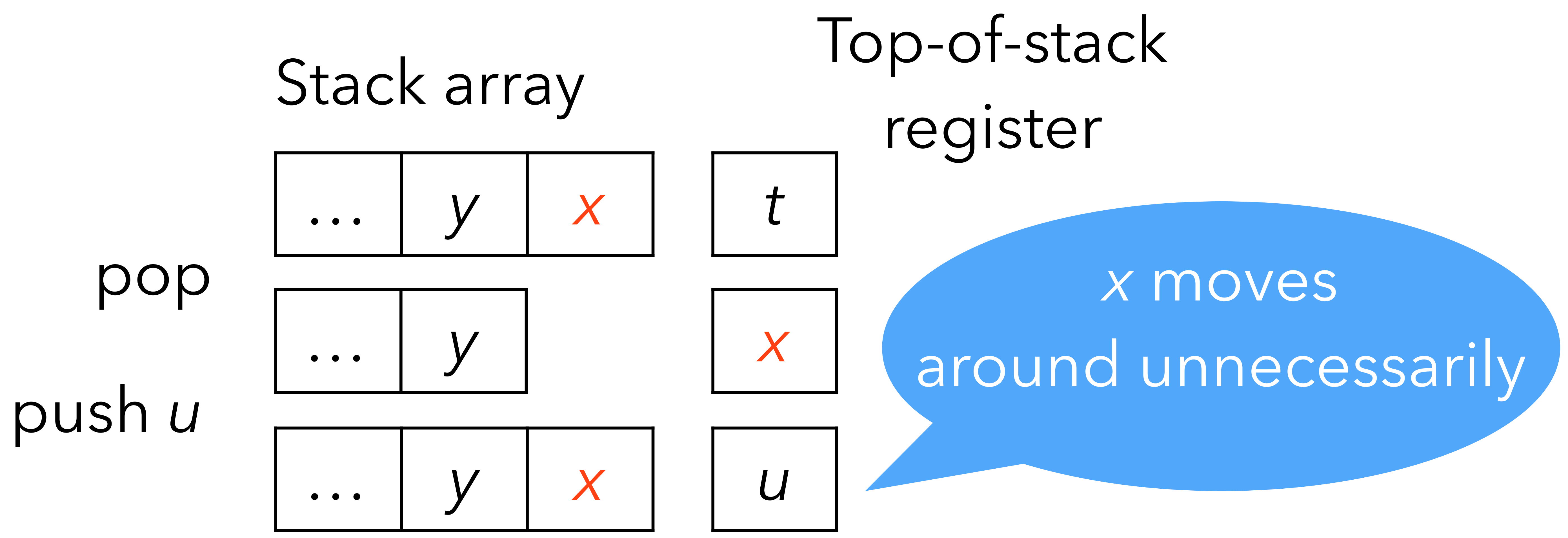
Top-of-stack caching

The top element is always cached:



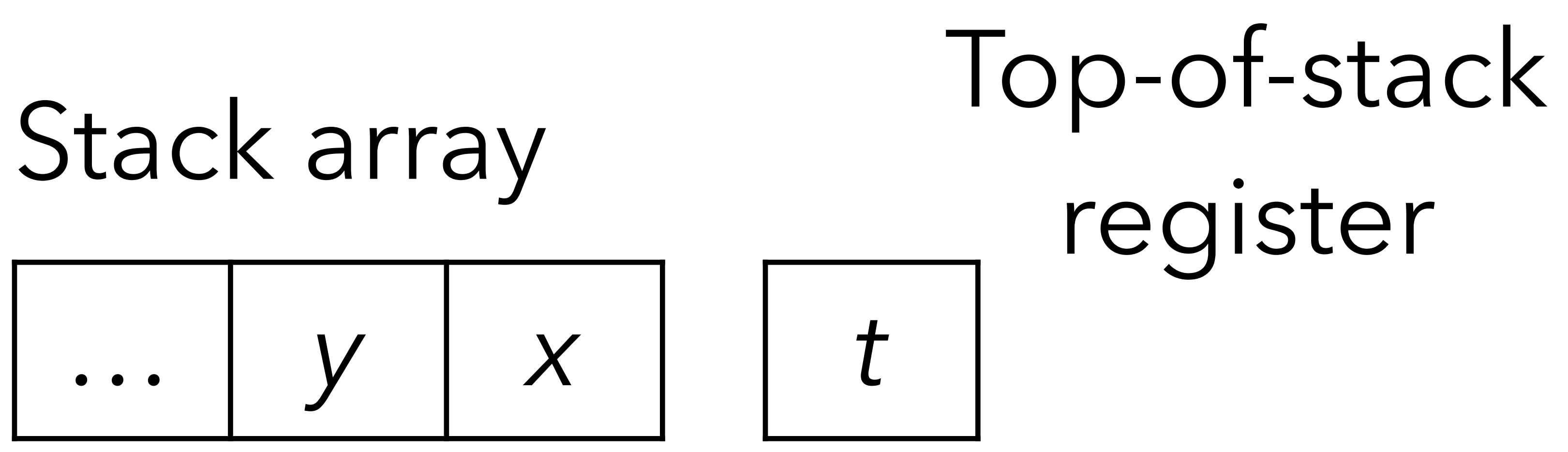
Top-of-stack caching

The top element is always cached:



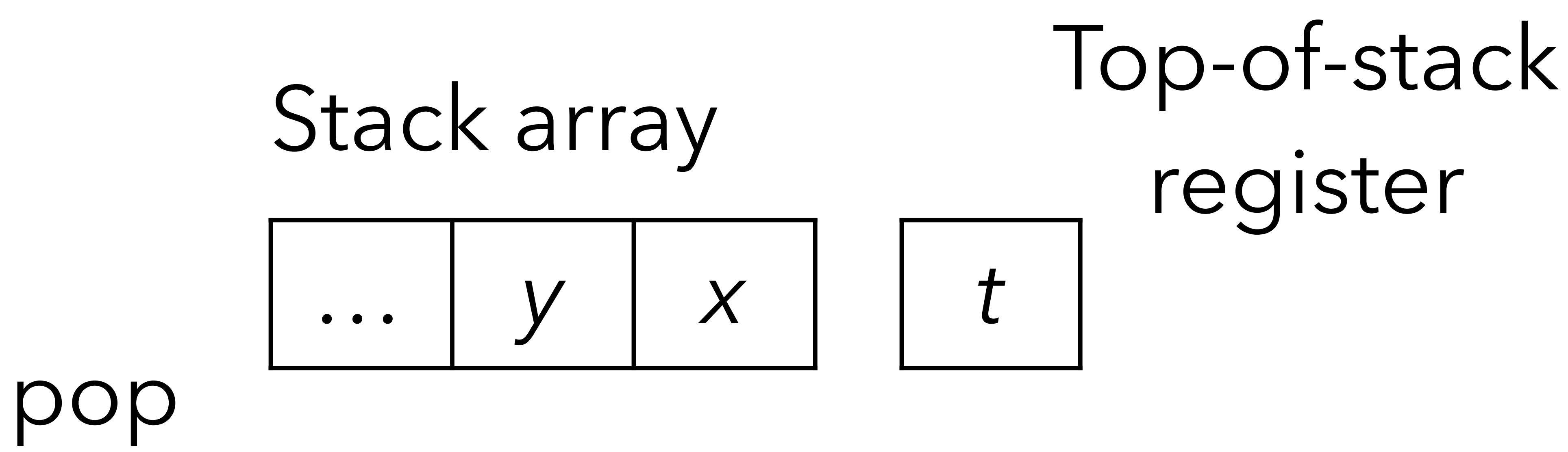
Top-of-stack caching

Either 0 or 1 top-of-stack element is cached:



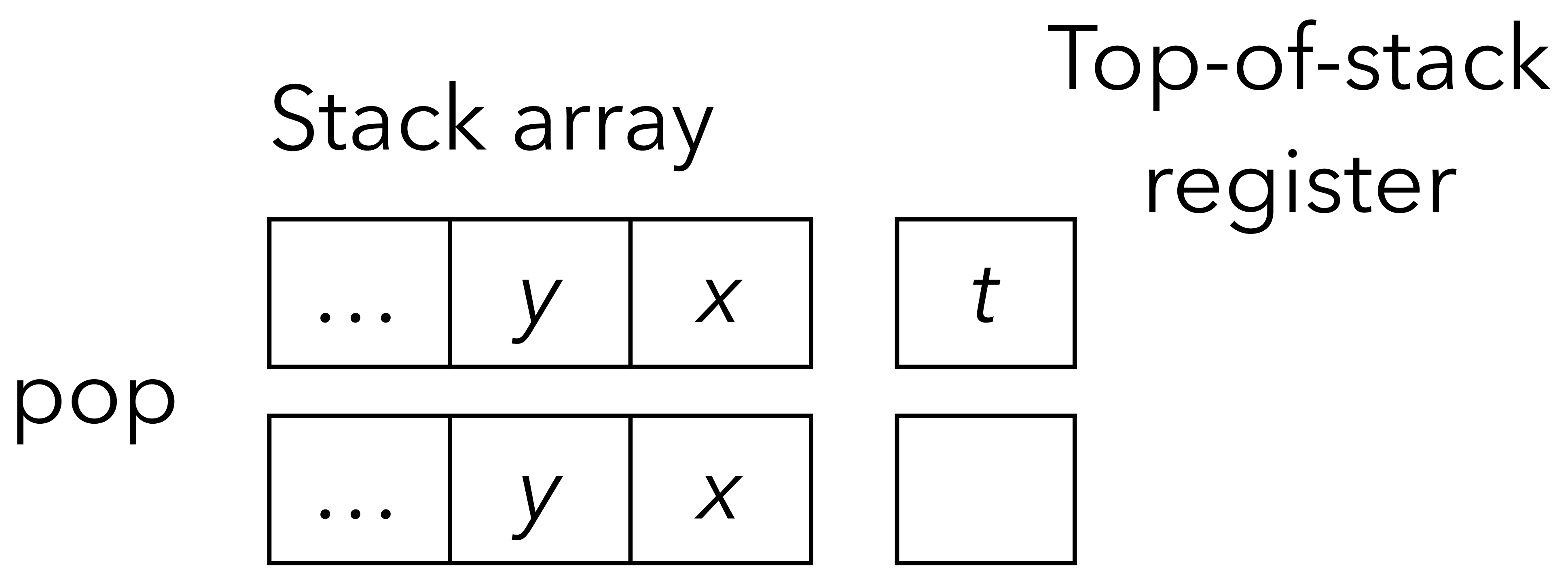
Top-of-stack caching

Either 0 or 1 top-of-stack element is cached:



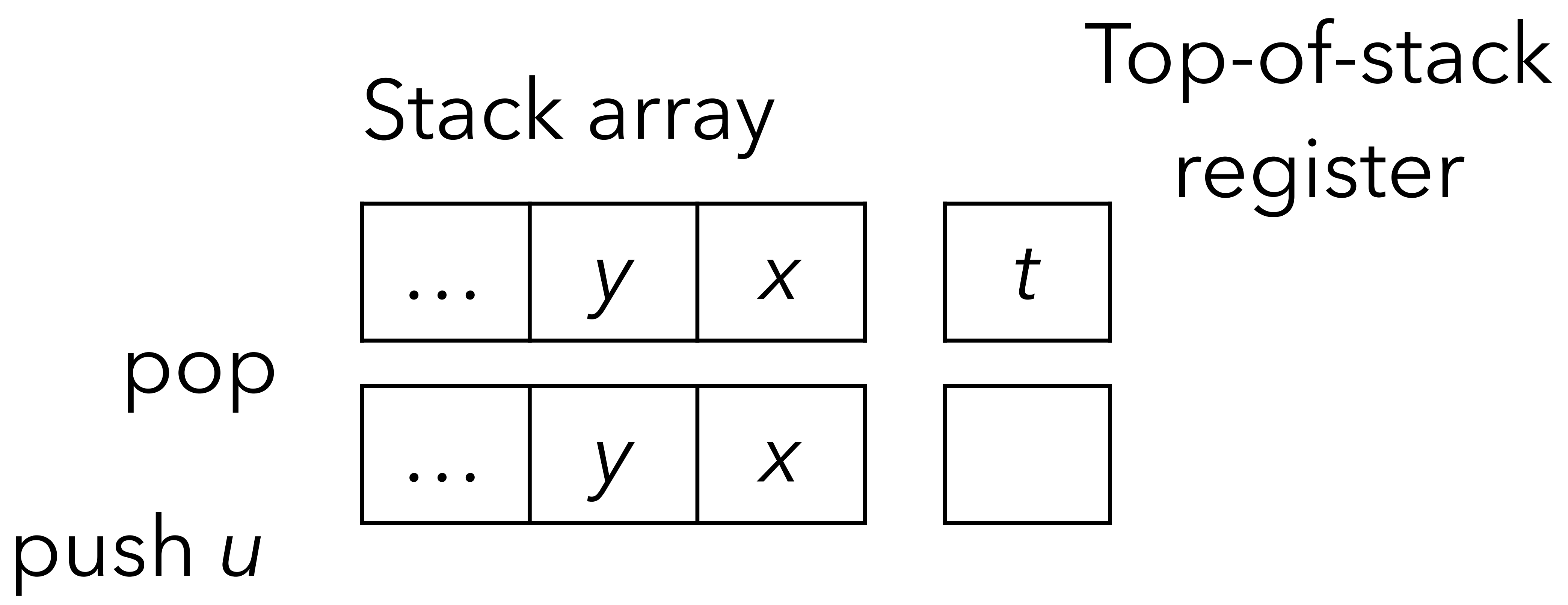
Top-of-stack caching

Either 0 or 1 top-of-stack element is cached:



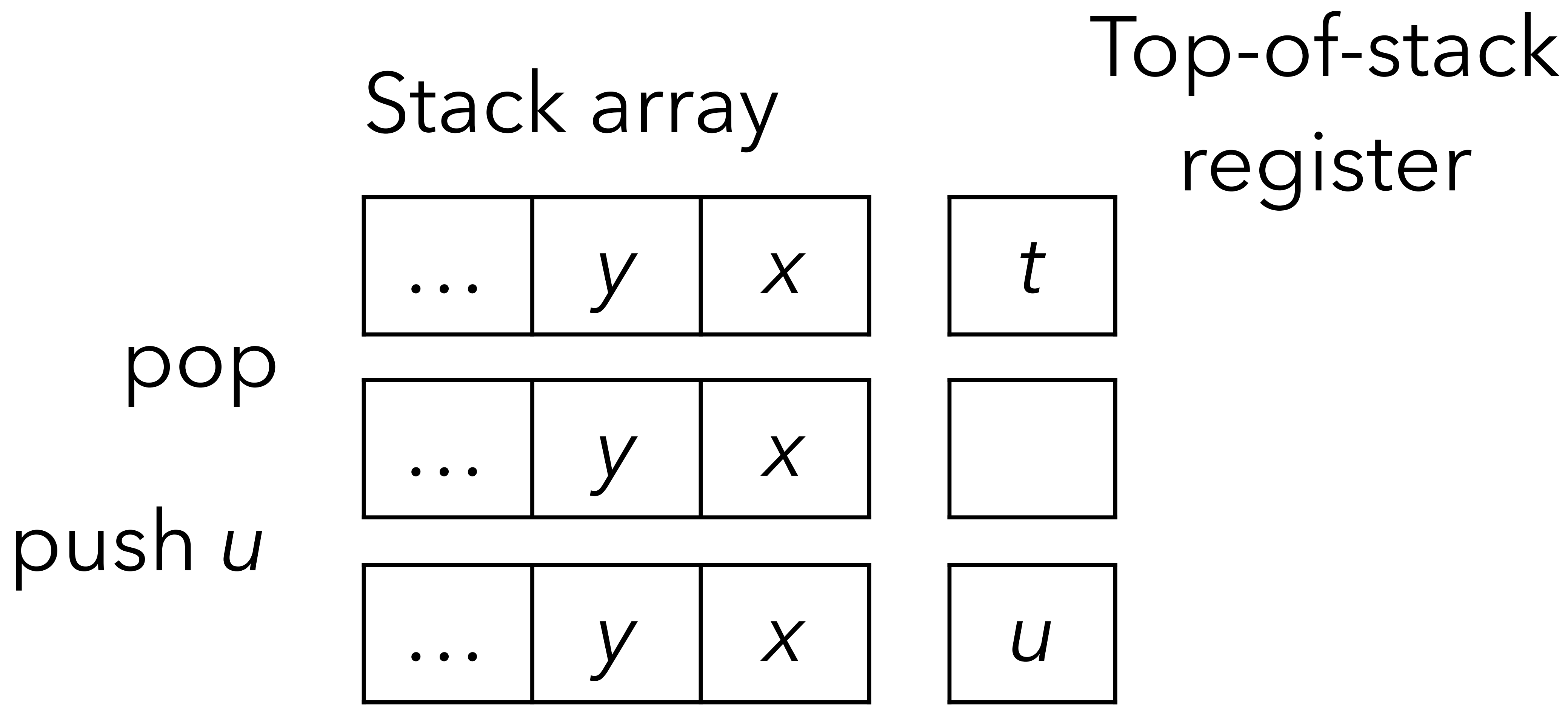
Top-of-stack caching

Either 0 or 1 top-of-stack element is cached:



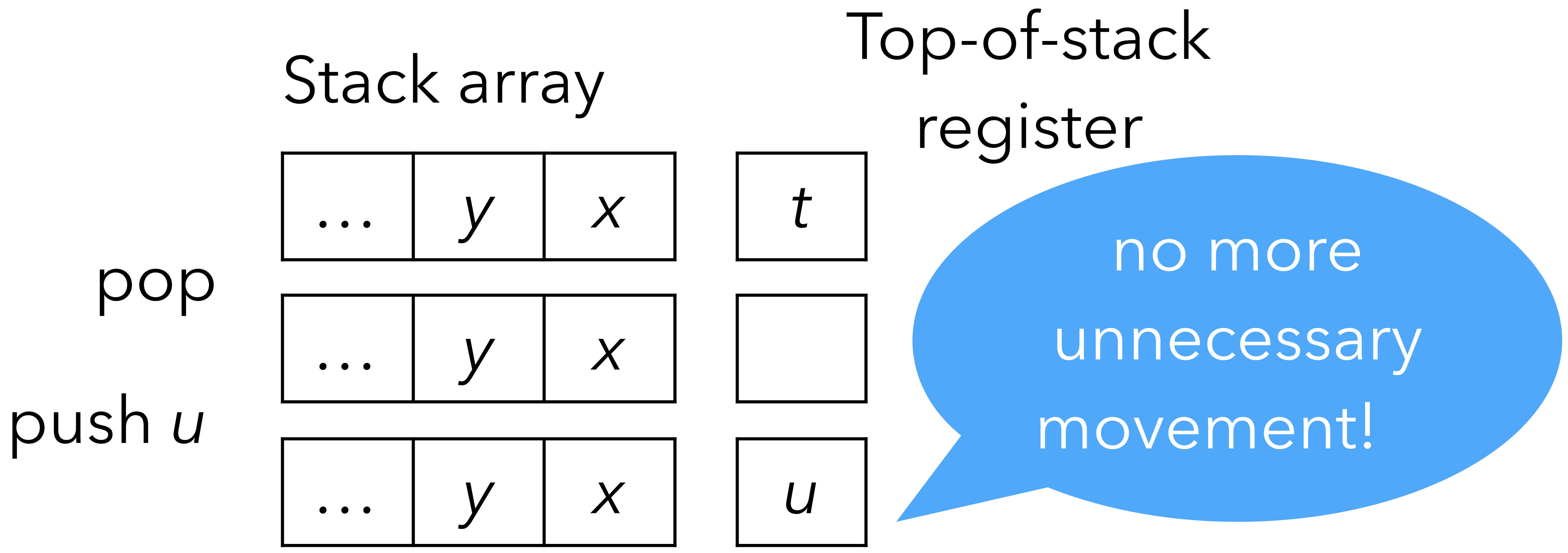
Top-of-stack caching

Either 0 or 1 top-of-stack element is cached:



Top-of-stack caching

Either 0 or 1 top-of-stack element is cached:



Top-of-stack caching

Beware: caching a variable number of stack elements means that every instruction must have one implementation per **cache state** (number of stack elements currently cached)

E.g., when caching at most one stack element, the `add` instruction needs the following two implementations:

State 0: no elements in reg.

```
add_0:
  tos = sp[0]+sp[1];
  sp += 2;
  // go to state 1
```

State 1: top-of-stack in reg.

```
add_1:
  tos += sp[0];
  sp += 1;
  // stay in state 1
```

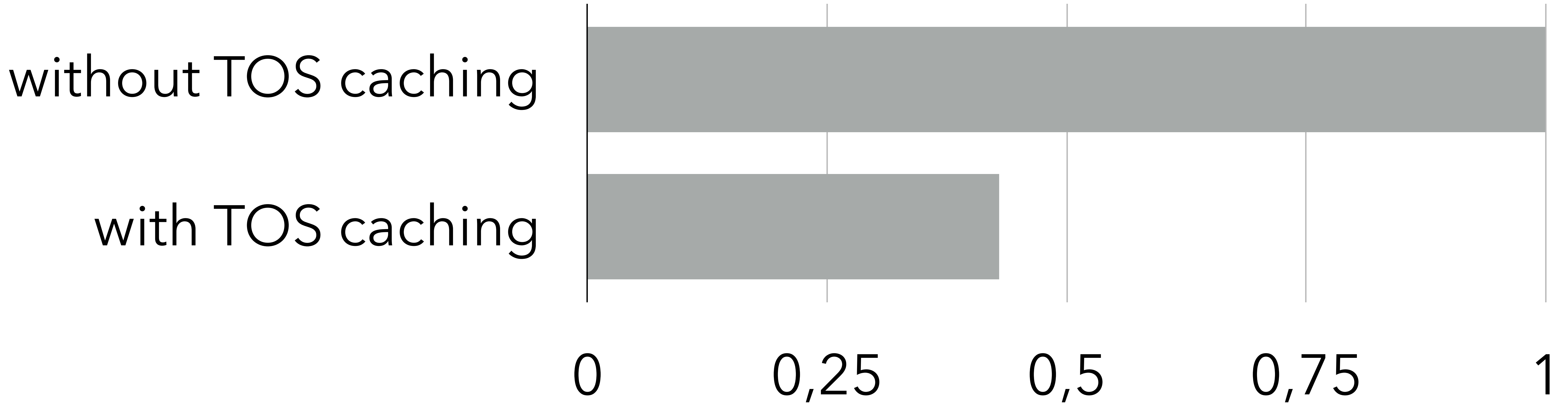
Benchmark

Benchmark: sum first 200'000'000 integers

Processor: 2.3 GHz Intel Core i9

Compiler: clang 11.0.3

Optimization settings: `-O3`



Super-instructions

Static super-instructions

Observation:

instruction dispatch is expensive in a VM.

Conclusion:

group several instructions into **super-instructions**.

Idea:

- use profiling to determine which sequences should be transformed into super-instructions,
- modify the the instruction set of the VM accordingly.

E.g., if `mul`, `add` appears often in sequence, combine the two in a single `madd` (multiply and add) super-instruction.

Dynamic super-instructions

Super-instructions can also be generated at run time, to adapt to the program being run.

This is the idea of **dynamic super-instructions**.

Pushed to its limits: generate one super-instruction per basic-block.

L3VM

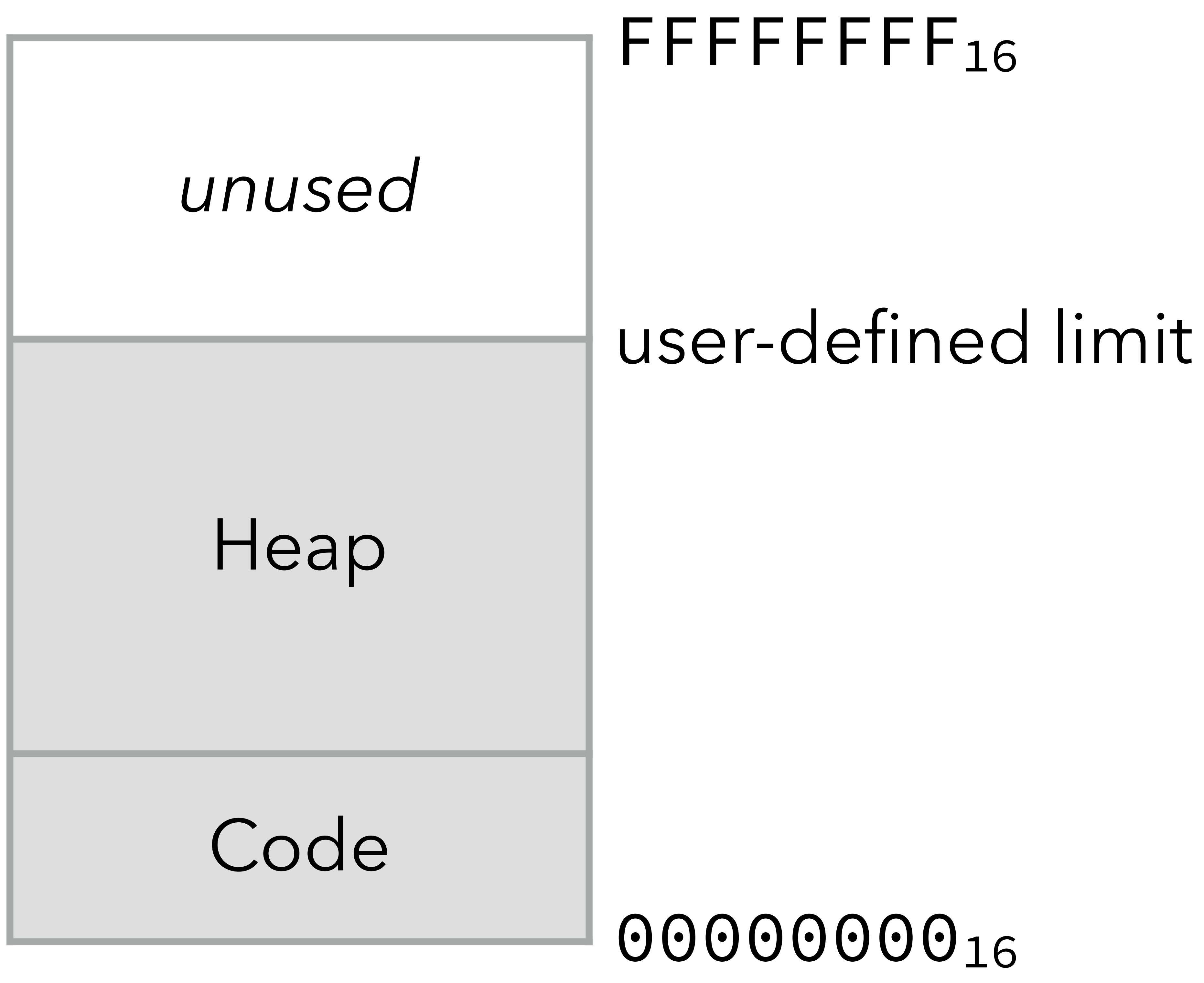
L_3VM

L_3VM is the VM of the L_3 project. Main characteristics:

- it is a 32 bits VM:
  - (untagged) integers are 32 bits,
  - pointers are 32 bits,
  - instructions are 32 bits,
- it is register-based (with an unconventional notion of register),
- it is simple: only 32 instructions.

Memory

Single 32-bit address space used to store code and heap.
Code is stored starting at address 0, the rest is used for the heap.
(Note: L_3VM addresses are not the same as those of the host).



Registers

Strictly speaking, L_3VM has only four registers:

- the **program counter** PC, which contains the address of the instruction being executed,
- the three **base registers** I_b, L_b and O_b, which contain either 0 or the address of a heap-allocated block.

(Pseudo-)registers

Base registers point to heap-allocated blocks, whose slots are the (pseudo-)registers used by the instructions. For example :

O_3 = slot at index 3 of block referenced by O_b.

There are:

- 32 **input** pseudo-registers (I_0 to I_{31}),
- 32 **output** pseudo-registers (O_0 to O_{31}),
- 160 **local** pseudo-registers (L_0 to L_{159}),

and:

- 32 **constant** pseudo-registers (C_0 to C_{31}) containing the constants 0 to 31.

Function call and return

In L_3VM, functions:

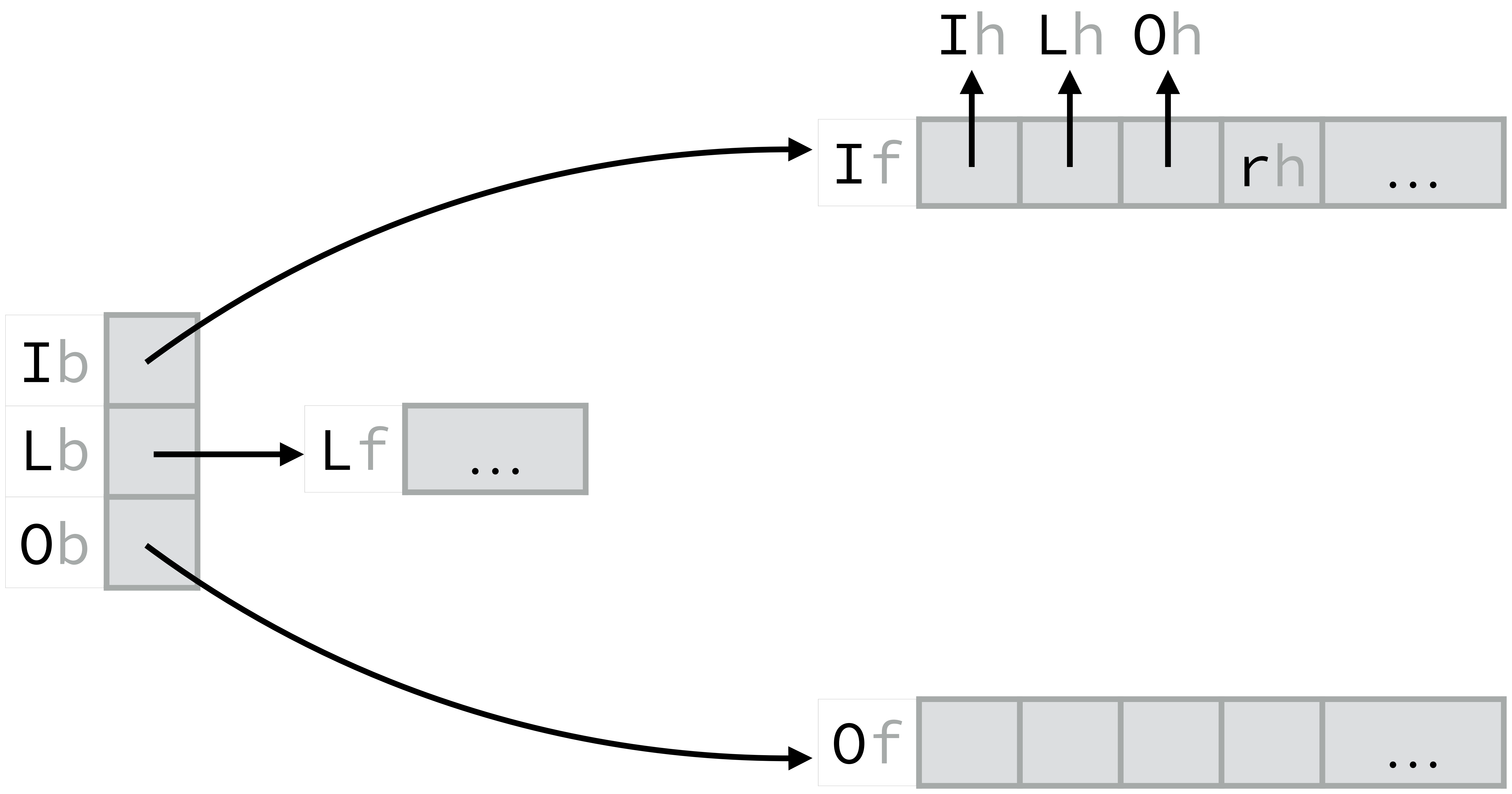
- get their arguments in their input registers (I_x),
- store their variables in their local registers (L_x),
- pass arguments to called functions through output registers (O_x).

To that end:

- `CALL_`… saves the caller's context (I_b, L_b, O_b and return address) in the callee's first four input registers,
- `RET` restores the caller's context.

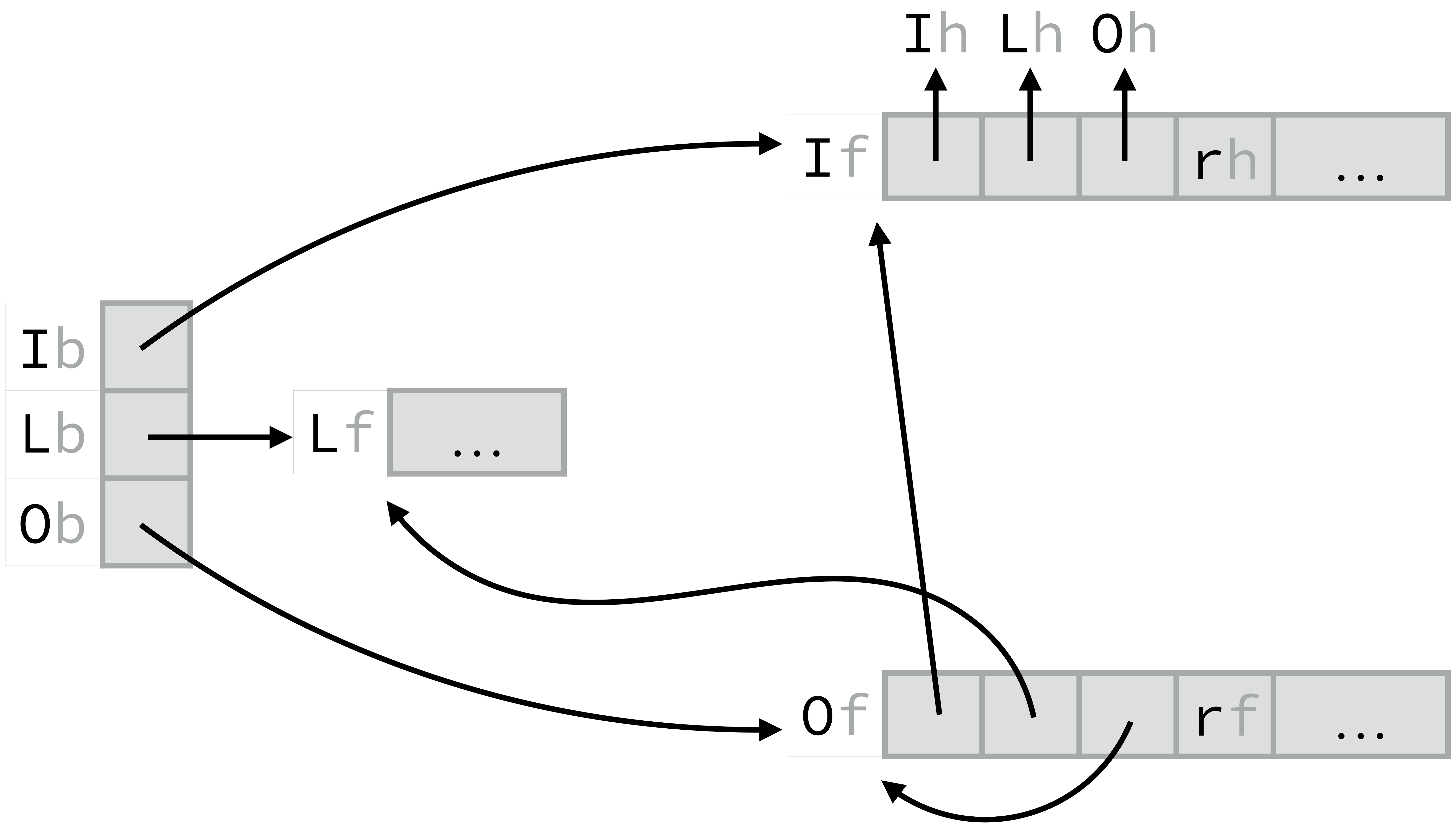
Non tail call example

Saving the caller's context and installing the callee's context during a non tail call from a function `f` to a function g, with h being `f`'s caller:



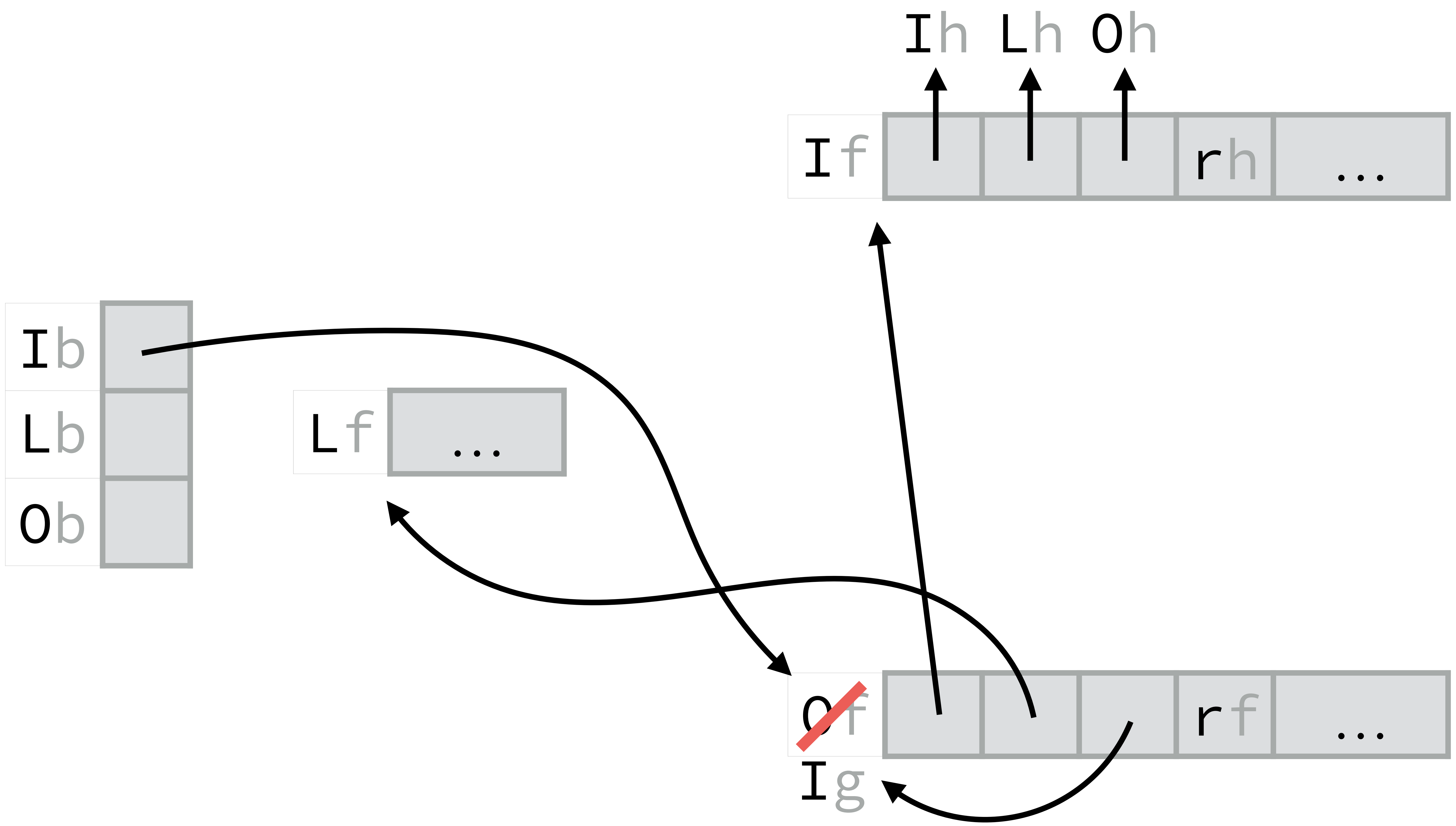
Non tail call example

Saving the caller's context and installing the callee's context during a non tail call from a function `f` to a function g, with h being `f`'s caller:



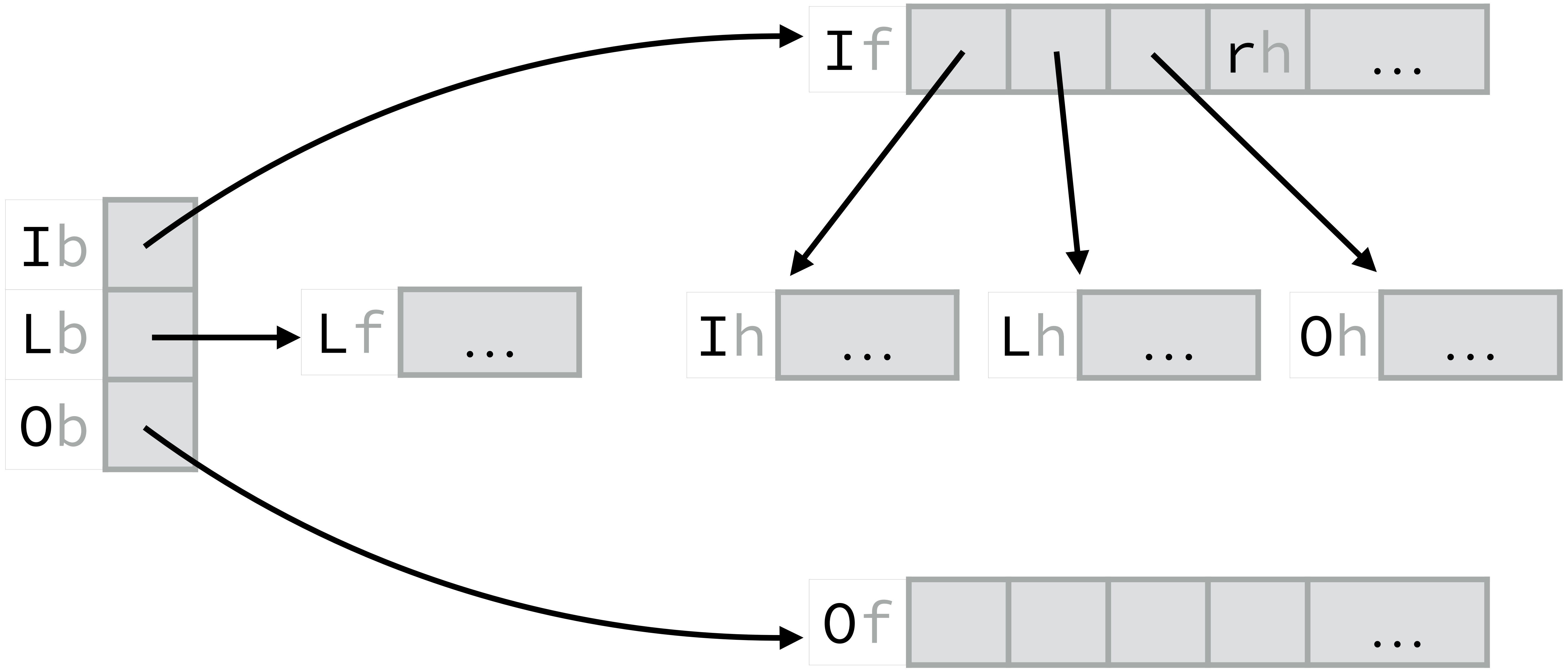
Non tail call example

Saving the caller's context and installing the callee's context during a non tail call from a function `f` to a function g, with h being `f`'s caller:



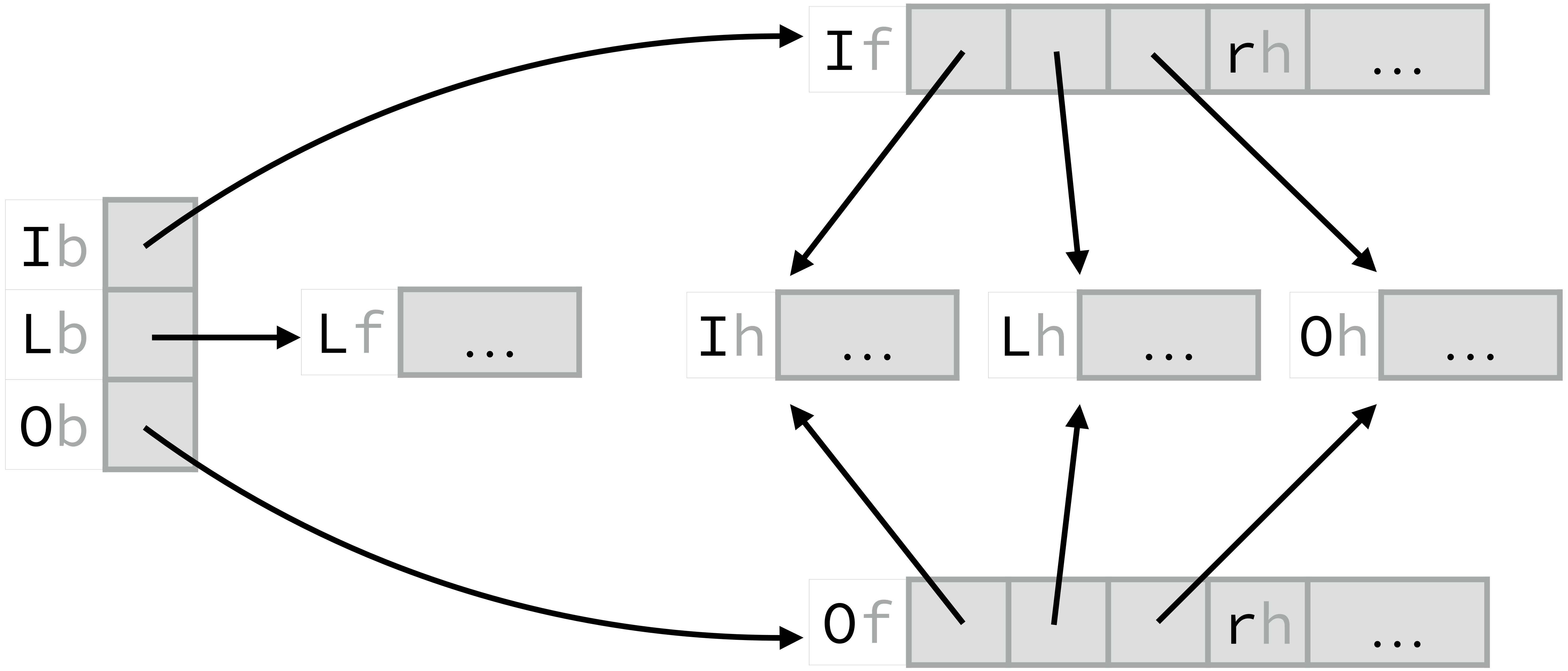
Tail call example

Saving the caller’s context and installing the callee’s context during a tail call from a function `f` to a function g, with `h` being `f`'s caller:



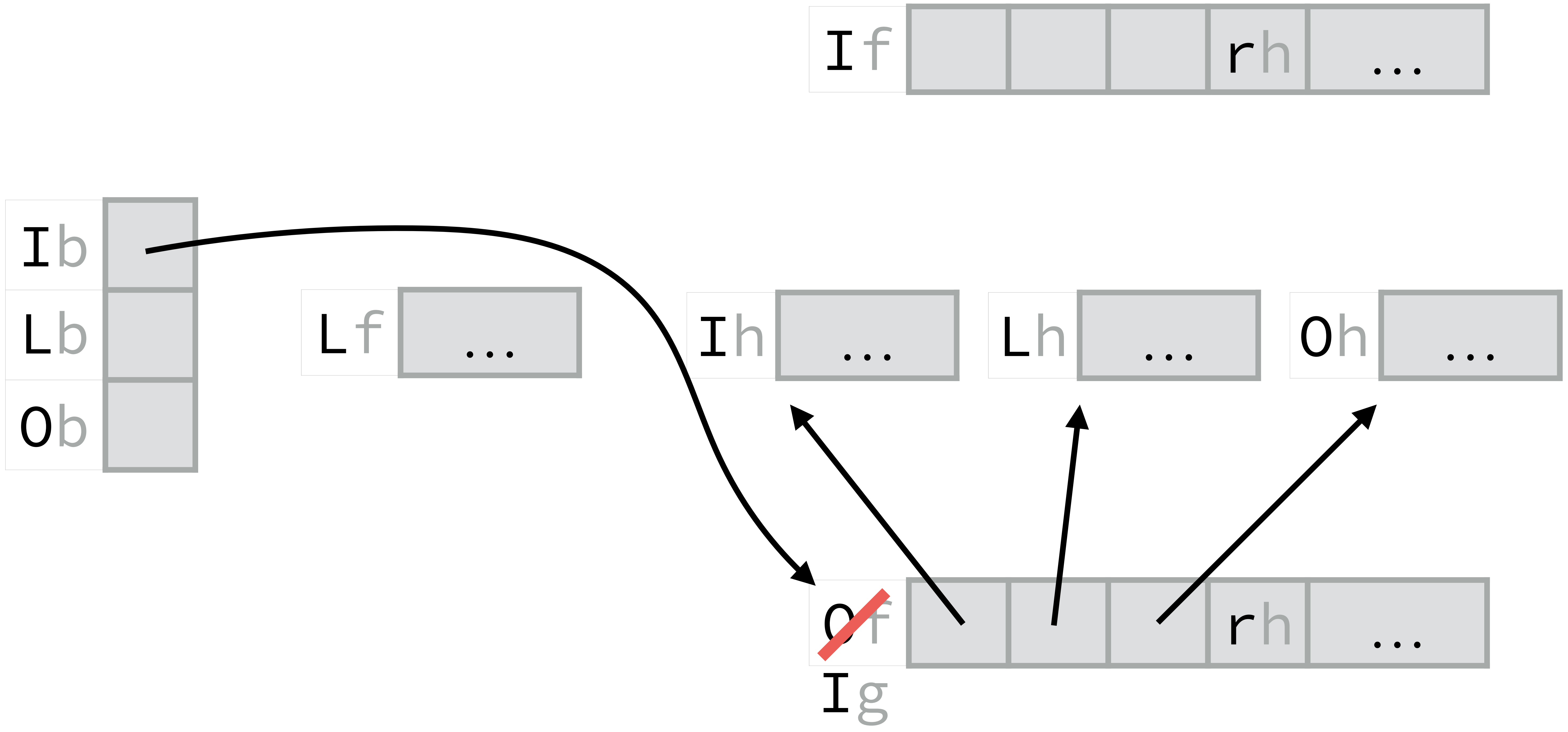
Tail call example

Saving the caller's context and installing the callee's context during a tail call from a function `f` to a function g, with `h` being `f`'s caller:



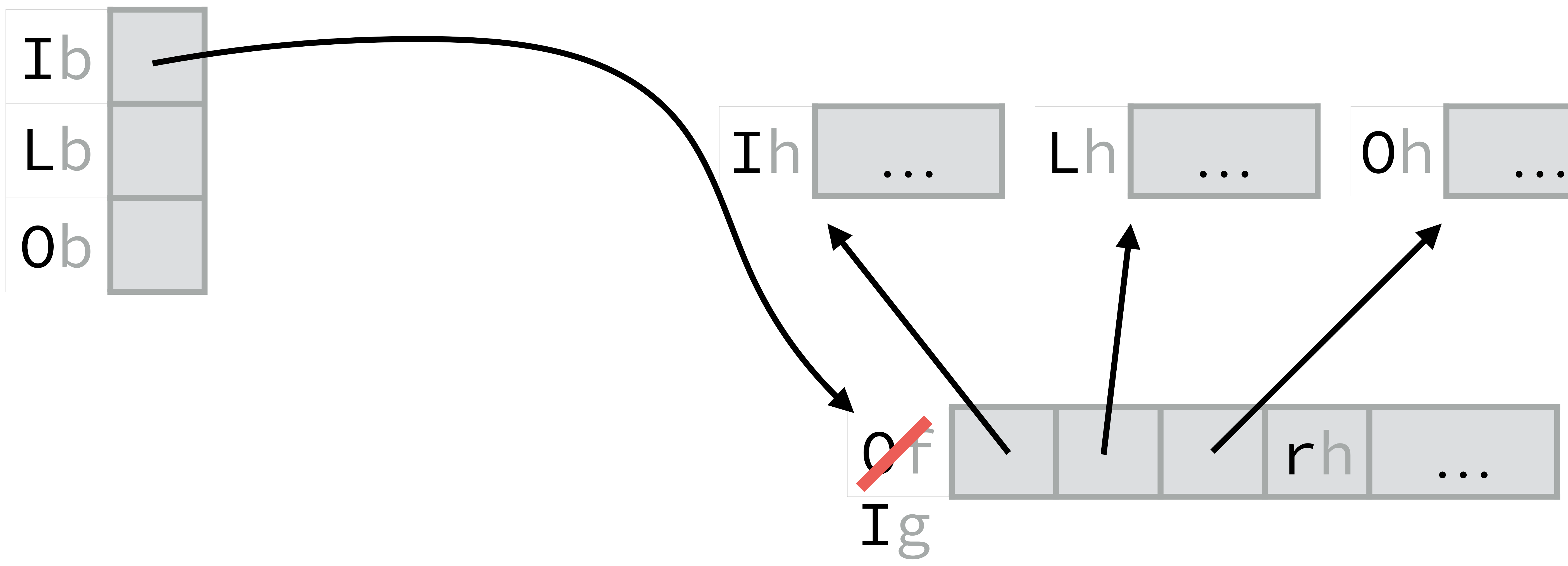
Tail call example

Saving the caller’s context and installing the callee’s context during a tail call from a function `f` to a function g, with h being `f`'s caller:



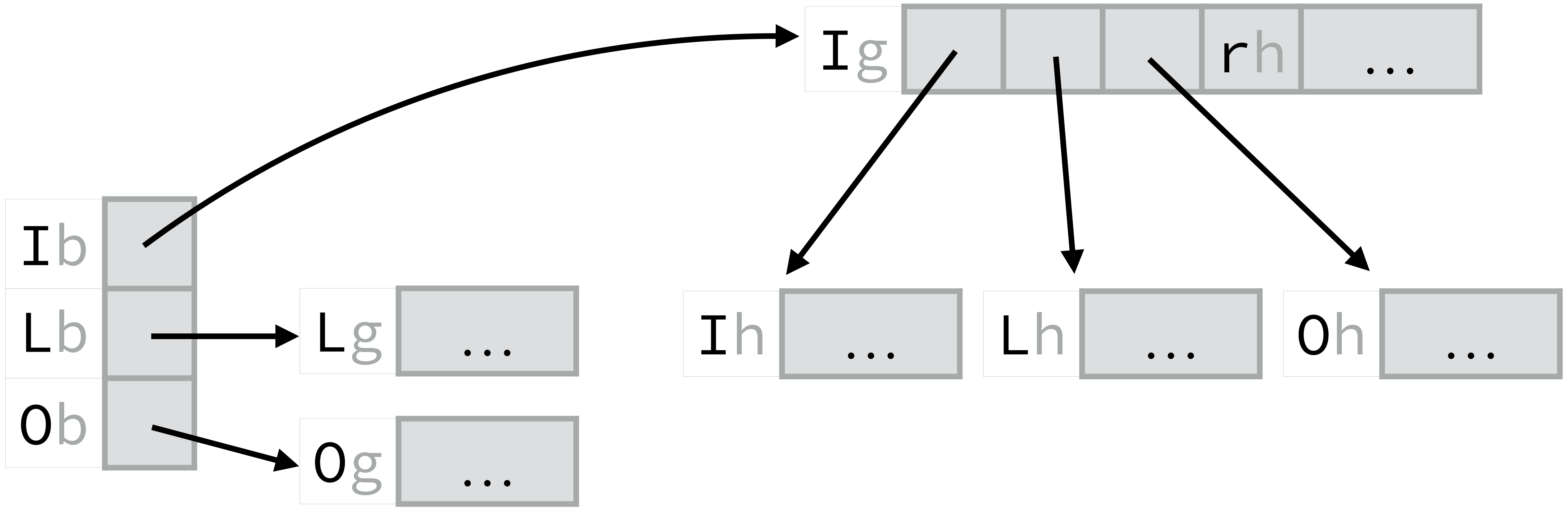
Tail call example

Saving the caller's context and installing the callee's context during a tail call from a function `f` to a function g, with h being `f`'s caller:



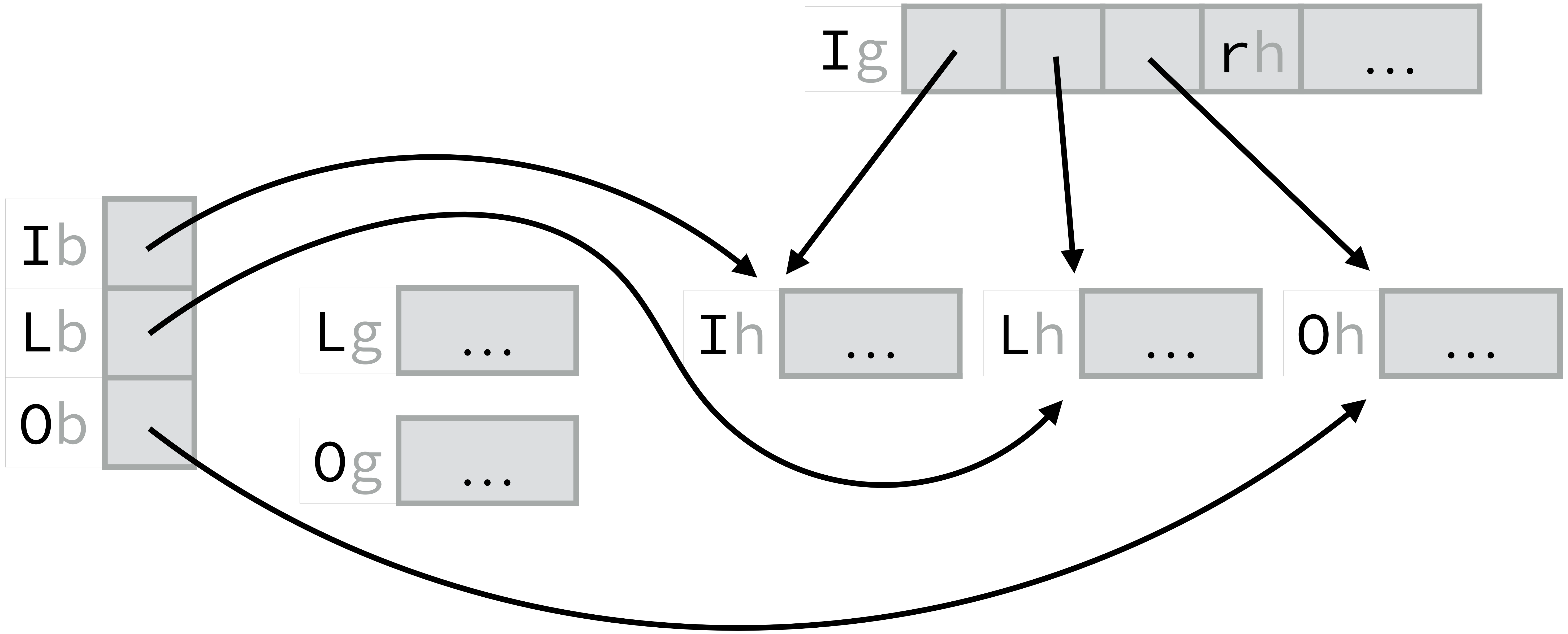
Return example

Restoring the caller's context during a function return from g to h (g was tail called from f) :



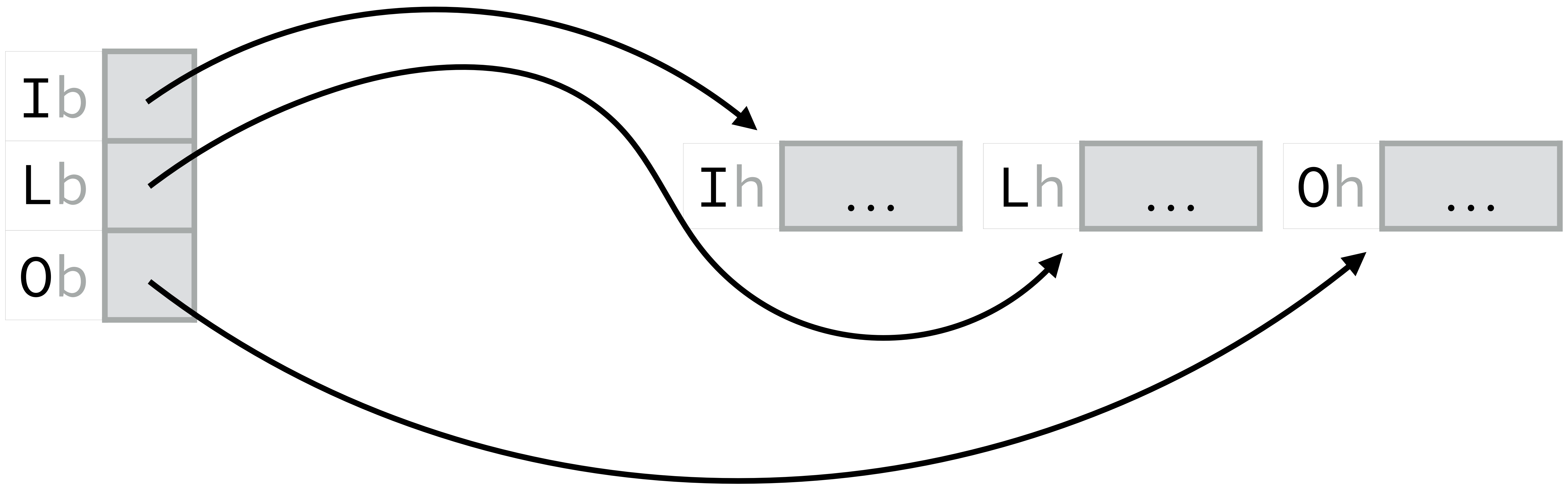
Return example

Restoring the caller's context during a function return from g to h (g was tail called from `f`) :



Return example

Restoring the caller's context during a function return from g to h (g was tail called from f) :



Arithmetic instructions (1)

`ADD` Ra Rb Rc	$Ra \leftarrow Rb + Rc$
`SUB` Ra Rb Rc	$Ra \leftarrow Rb - Rc$
`MUL` Ra Rb Rc	$Ra \leftarrow Rb \times Rc$
`DIV` Ra Rb Rc	$Ra \leftarrow Rb / Rc$
`MOD` Ra Rb Rc	$Ra \leftarrow Rb \% Rc$

Ra, Rb, Rc: registers

PC implicitly augmented by 4 by each instruction

Arithmetic instructions (2)

LSL *Ra* *Rb* *Rc*	$Ra \leftarrow Rb << Rc$
LSR *Ra* *Rb* *Rc*	$Ra \leftarrow Rb >> Rc$
AND *Ra* *Rb* *Rc*	$Ra \leftarrow Rb \mathbin{\&} Rc$
OR *Ra* *Rb* *Rc*	$Ra \leftarrow Rb \mid Rc$
XOR *Ra* *Rb* *Rc*	$Ra \leftarrow Rb \wedge Rc$

Ra, *Rb*, *Rc*: registers
PC implicitly augmented by 4 by each instruction

Control instructions (1)

`JLT` $Ra\ Rb\ D^{11}$	if $Ra < Rb$ then PC ← PC + $4 \cdot D^{11}$
`JLE` $Ra\ Rb\ D^{11}$	if $Ra \leq Rb$ then PC ← PC + $4 \cdot D^{11}$
`JEQ` $Ra\ Rb\ D^{11}$	if $Ra = Rb$ then PC ← PC + $4 \cdot D^{11}$
`JNE` $Ra\ Rb\ D^{11}$	if $Ra \neq Rb$ then PC ← PC + $4 \cdot D^{11}$
`JI` D^{27}	PC ← PC + $4 \cdot D^{27}$

Ra, *Rb*, *Rc*: registers,
D^k: *k*-bit signed displacement

Control instructions (2)

CALL_NI *Ra*	$(O_0, O_1, O_2, O_3) \leftarrow (I_b, L_b, O_b, PC + 4)$, $I_b \leftarrow O_b$, $PC \leftarrow Ra$
CALL_ND D^{27}	*like* CALL_NI, *except that* $PC \leftarrow PC + 4 \cdot D^{27}$
CALL_TI *Ra*	$(O_0, O_1, O_2, O_3) \leftarrow (I_0, I_1, I_2, I_3)$, $I_b \leftarrow O_b$, $PC \leftarrow Ra$
CALL_TD D^{27}	*like* CALL_TI, *except that* $PC \leftarrow PC + 4 \cdot D^{27}$
RET *Ra*	$r \leftarrow Ra$, $(PC, O_b, L_b, I_b) \leftarrow (I_3, I_2, I_1, I_0)$, $O_0 \leftarrow r$
HALT *Ra*	halt execution with the value of *Ra*

Ra: register,
D^k: *k*-bit signed displacement,
r: temporary value

Register instructions

`LDLO` Ra, S^{19}	$Ra \leftarrow S^{19}$
`LDHI` Ra, U^{16}	$Ra \leftarrow (U^{16} \ll 16) \mid (Ra \ \& \ \mathrm{FFFF}_{16})$
`MOVE` Ra, Rb	$Ra \leftarrow Rb$
`RALO` U^8, V^8	$\mathrm{L_b} \leftarrow$ new block of size U^8 and tag 201 $\mathrm{O_b} \leftarrow$ new block of size V^8 and tag 201

Ra, Rb: registers,
S^k: k-bit signed constant,
U^k, V^k: k-bit unsigned constants
PC implicitly augmented by 4 by each instruction

Block instructions

`BALO` Ra Rb T^8	$Ra \leftarrow$ new block of size Rb and tag T^8
`BSIZ` Ra Rb	$Ra \leftarrow$ size of block Rb
`BTAG` Ra Rb	$Ra \leftarrow$ tag of block Rb
`BGET` Ra Rb Rc	$Ra \leftarrow$ element at index Rc of block Rb
`BSET` Ra Rb Rc	element at index Rc of block $Rb \leftarrow Ra$

Ra, Rb, Rc: registers,
T^8: 8-bit block tag
PC implicitly augmented by 4 by each instruction

I/O instructions

`BREA` *Ra*	*Ra* ← byte read from console
`BWRI` *Ra*	write least-significant byte of *Ra* to console

Ra: register

PC implicitly augmented by 4 by each instruction

Example

The factorial in (hand-coded) L_3VM assembly:

```
        ;; I₄ contains argument
        ;; O₀ contains return value (after call)
fact:   RALO 0,5
        JNE C₀,I₄,else
        RET C₁
else:   SUB O₄,I₄,C₁
        CALL_ND fact
        MUL I₄,I₄,O₀
        RET I₄
```