VMs for Dynamic Languages

© 2015 Oracle, Mario Wolczko

Challenges of dynamic languages

- Types are unknown ahead of time. language itself does not involve any type information.
- Types may vary over time
- Intermingling of structured and primitive values
- Varying code (reflection, eval)
  - Programmer changes
  - Self-modifying code

`function f(a,b) { return a+b;}`

- Types are unknown ahead of time `f(x,y);`

- Types may vary over time
  `f(1, 2); … f(1.1, 2.7);`

- Intermingling of structured and primitive values
  `f(1,2); … f("hello ", "world");`

- Varying code (reflection, eval)
  `eval("f(1,2)");`
  `eval("function f(a,b){return a-b;}");`

take a string constant code and execute it.

Absence of type information on **data**

- No type information on variables, but values carry their own type
- When allocating space for a variable, must allocate for the biggest case, or more typically use an indirection for all values (see later)

Variable's type can not be determined ahead of time,

Ensuring their correctness. { ① allocate the maximum memory. so that it can hold any type of data
② using indirect lookup.

Example: expression bytecode interpreter, mixed mode

- Assume any variable can hold an int or float at any time. Bytecodes are untyped. Expressions involving a float use float arithmetic.

Expression bytecode interpreter

In this language, can allocate for the biggest case.

```
typedef struct {
  enum { INT, FLOAT } tag;
  union {
    int intVal;
    float floatVal;
  } datum;
} Val;

Val stack[MAXSTACK];
Val vars[MAXVARS];
Val *sp= &stack[-1];  // empty to begin
```

→ which type (not complicated)

Expression bytecode interpreter

```
case o(LITINT): ++sp; if (stack_depth() >= MAXSTACK) error();
                sp->tag= INT;  sp->intVal= get4int(pc+1);
                pc += 5;  continue;
case o(ADD):   if (stack_depth() < 2) error();
               if (sp[0].tag==INT)
                 if (sp[-1].tag==INT)  // both ints
                   sp[-1].intVal += sp[0].intVal;
                 else  // sp[-1] float, sp[0] int
                   sp[-1].floatVal += (float)(sp[0].intVal);
               else
                 if (sp[-1].tag==FLT)  // both floats
                   sp[-1].floatVal += sp[0].floatVal;
                 else {  // sp[-1] int, sp[0] float
                   sp[-1].tag= FLOAT;
                   sp[-1].floatVal= (float)(sp[-1].intVal)
                                  + sp[0].floatVal;
                 }
               --sp; ++pc;
               continue;
case o(GET)   : *++sp= vars[L5(*pc)]; // struct copy
               ++pc; continue;
```

trait forward & clean

Abstract interpretation of untyped bytecode

Cf. statically typed bytecode from earlier section.

```
case o(LITINT): push(INT); inc_pc(5); continue;
case o(LITFLT): push(FLT); inc_pc(5); continue;
case o(GET): push(vars[L(*pc)].type); ++pc; continue;
case o(PUT): if (stack_empty()) error();
             vars[L(*pc)]= pop(); ++pc; continue;
case o(ADD): if (stack_depth()<2) error();
             res= tos()==FLT || nos()==FLT ? FLT : INT;
             pop(); tos()= res; ++pc; continue;
```

Need type system to support.

Absence of type information on large data

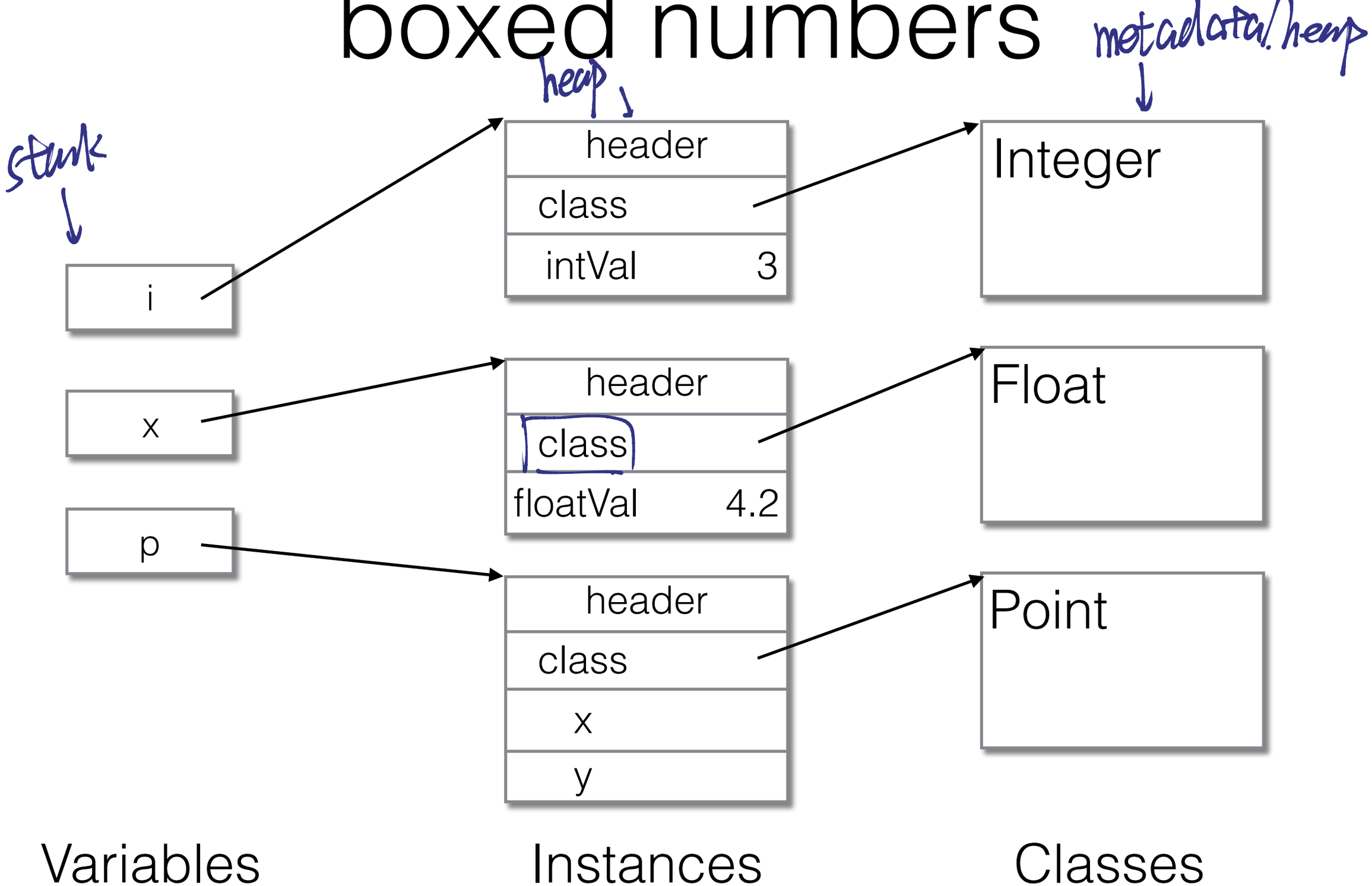
- When worst case allocation is impractical (e.g., a variable could reference a boolean or a million-element array), use an indirection to reference the value.
  - Allows for sharing (i.e., copying of references)
  - Variable contains an address or an index into an indirection table (aka object table)

Example

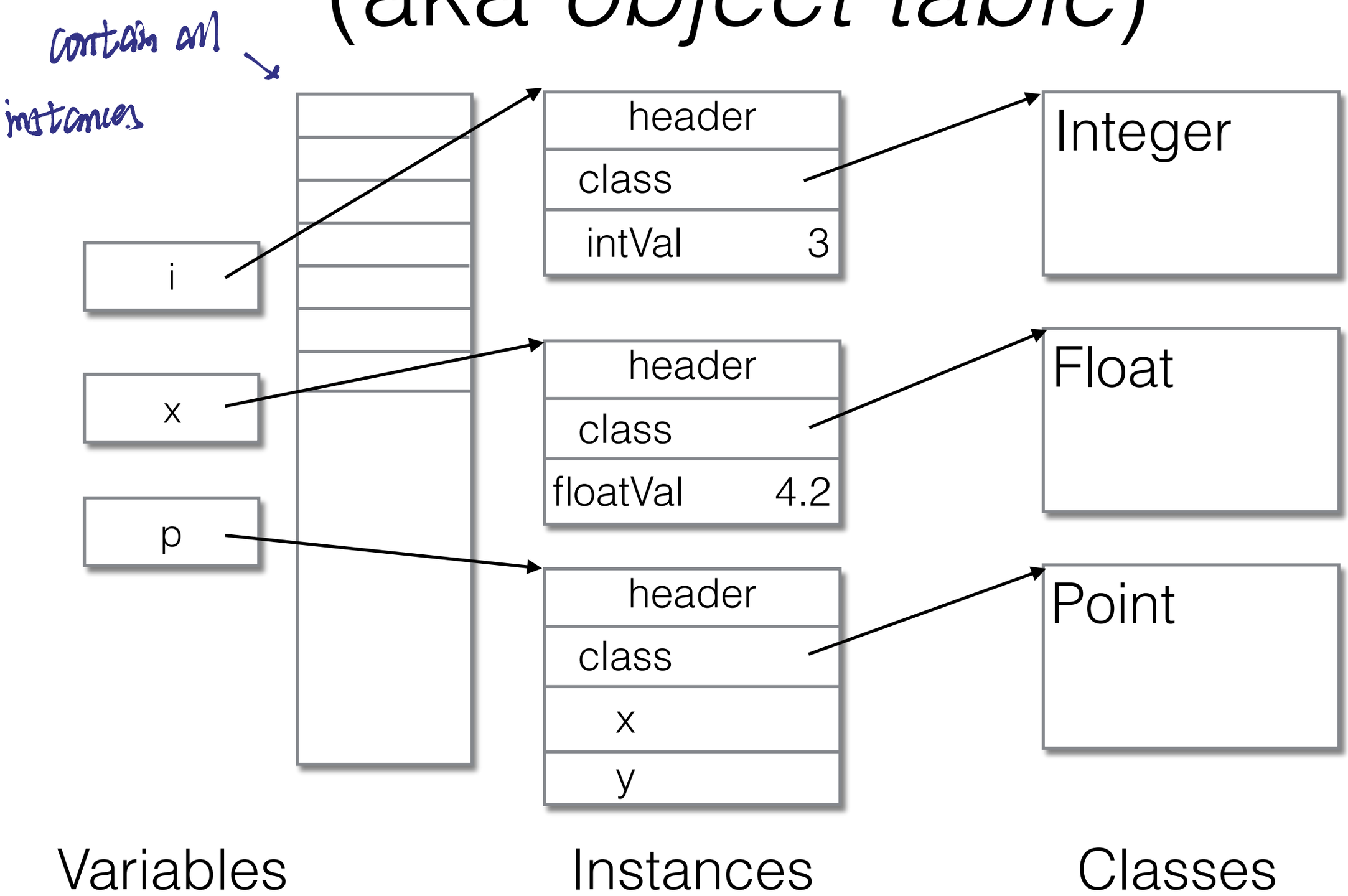
```
i = 3;

x = 4.2;

p = new Point(3, 4.2);
```

As long as we get the instance pointer. We can get the type of class instantly.

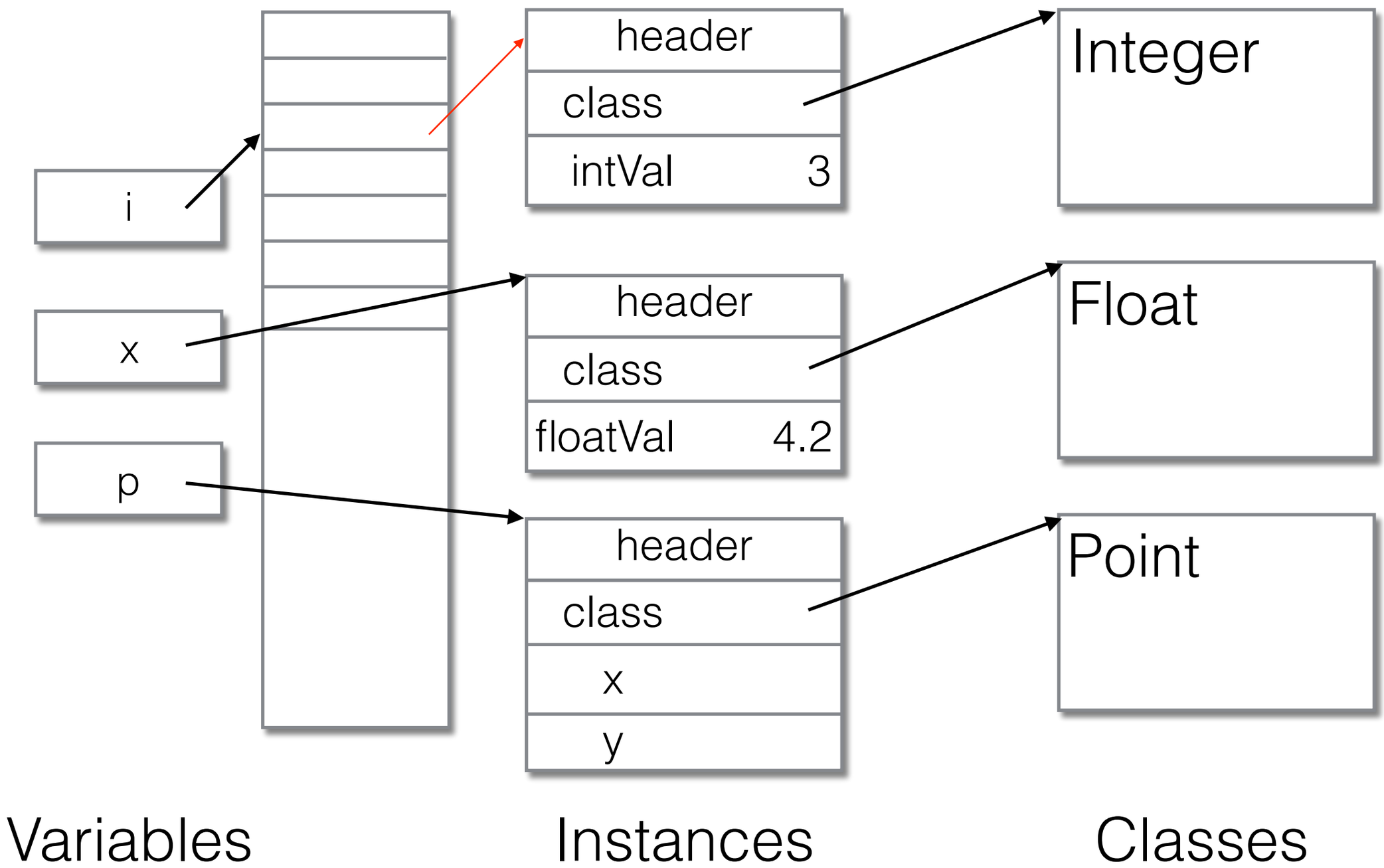
Direct heap organization, boxed numbers



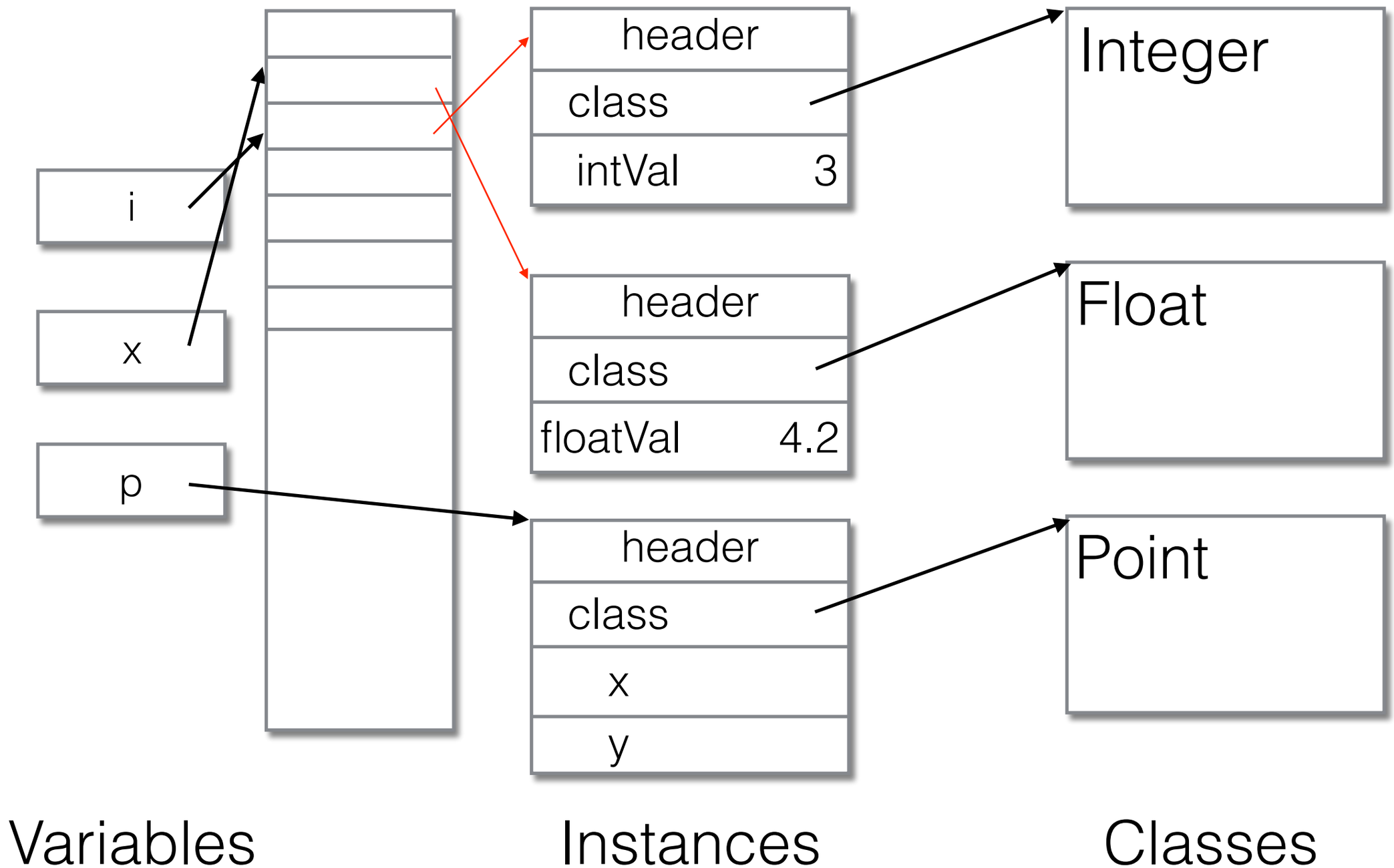
Indirect heap organization (aka *object table*)



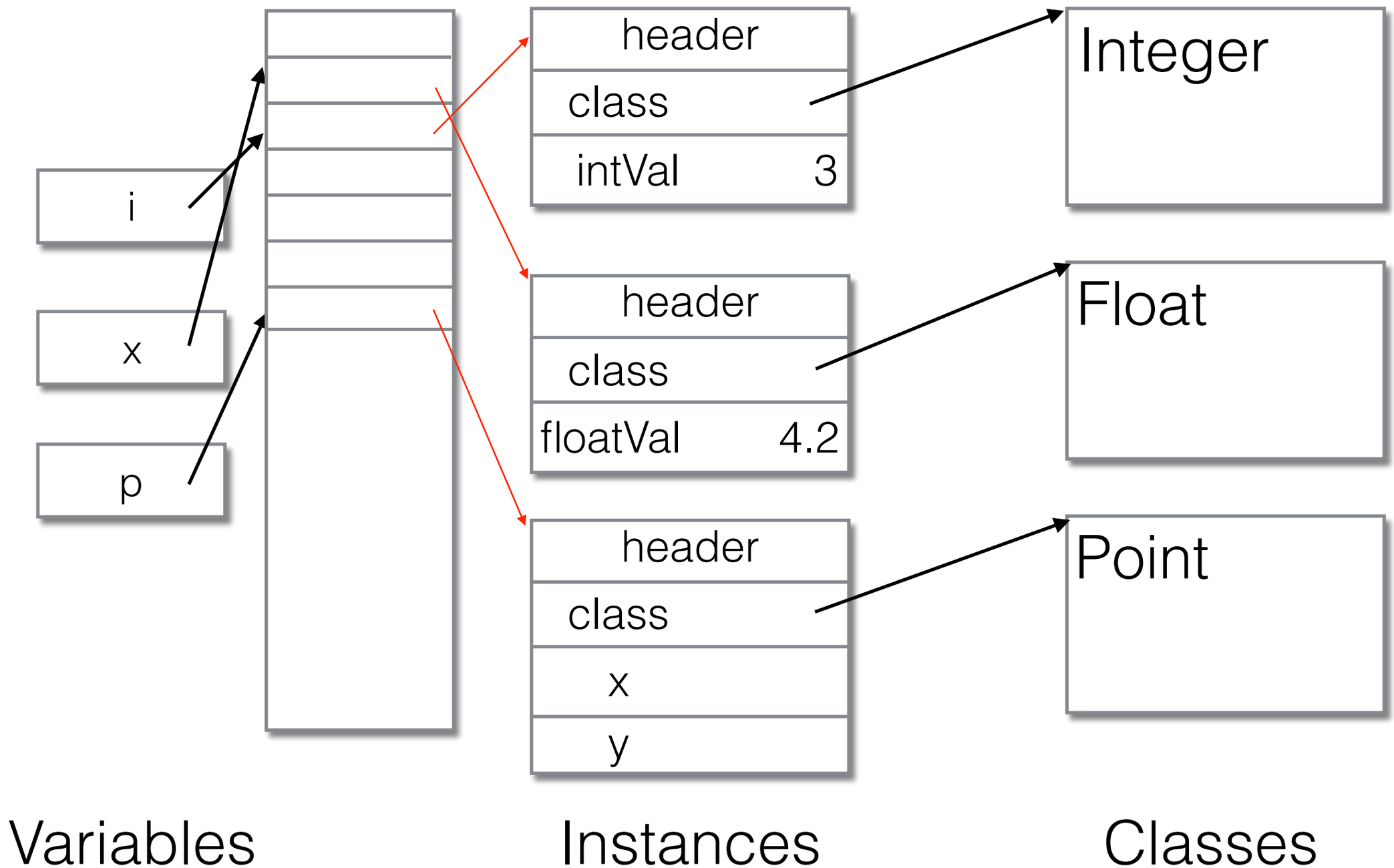
Indirect heap organization (aka *object table*)



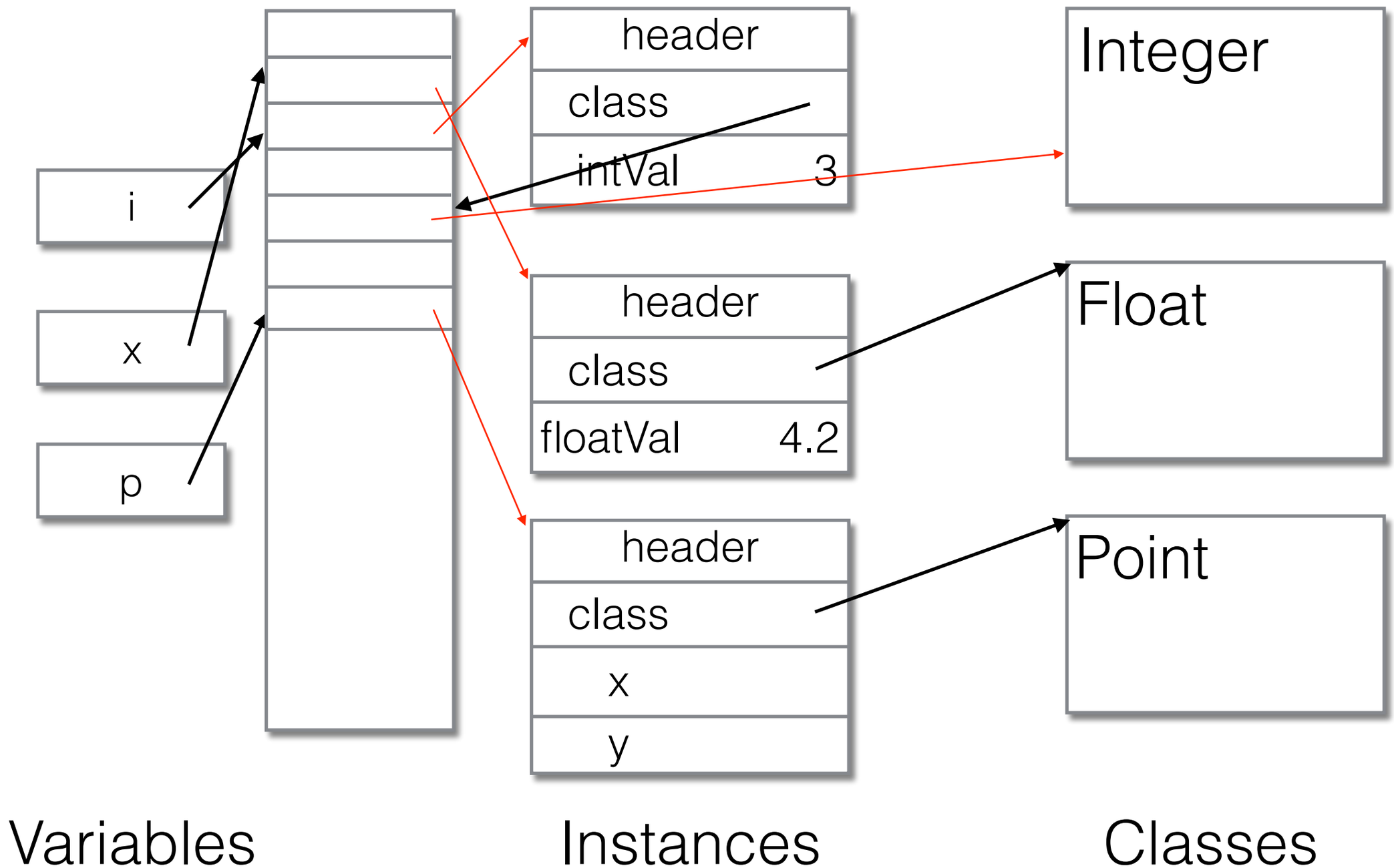
Indirect heap organization (aka *object table*)



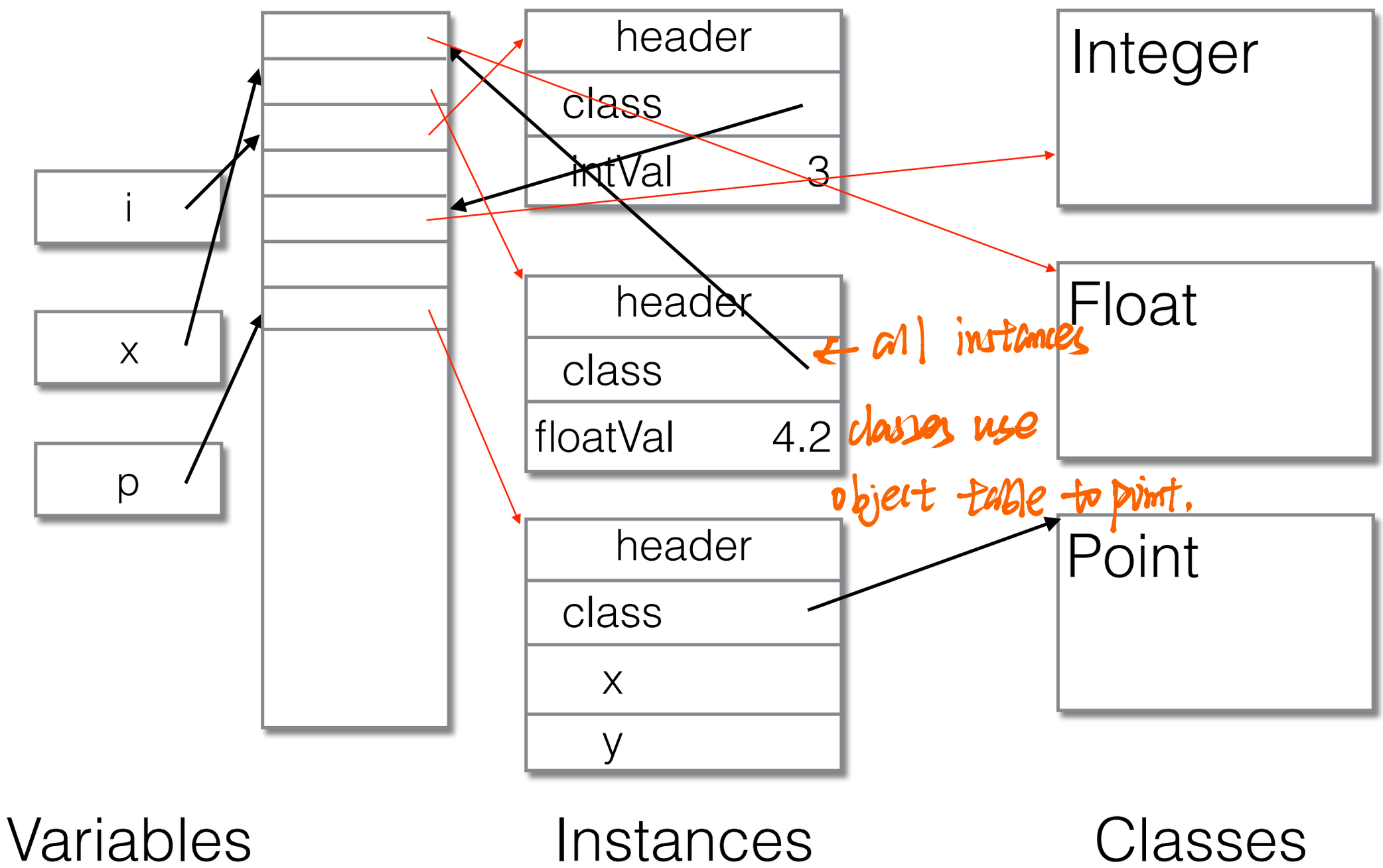
Indirect heap organization (aka *object table*)



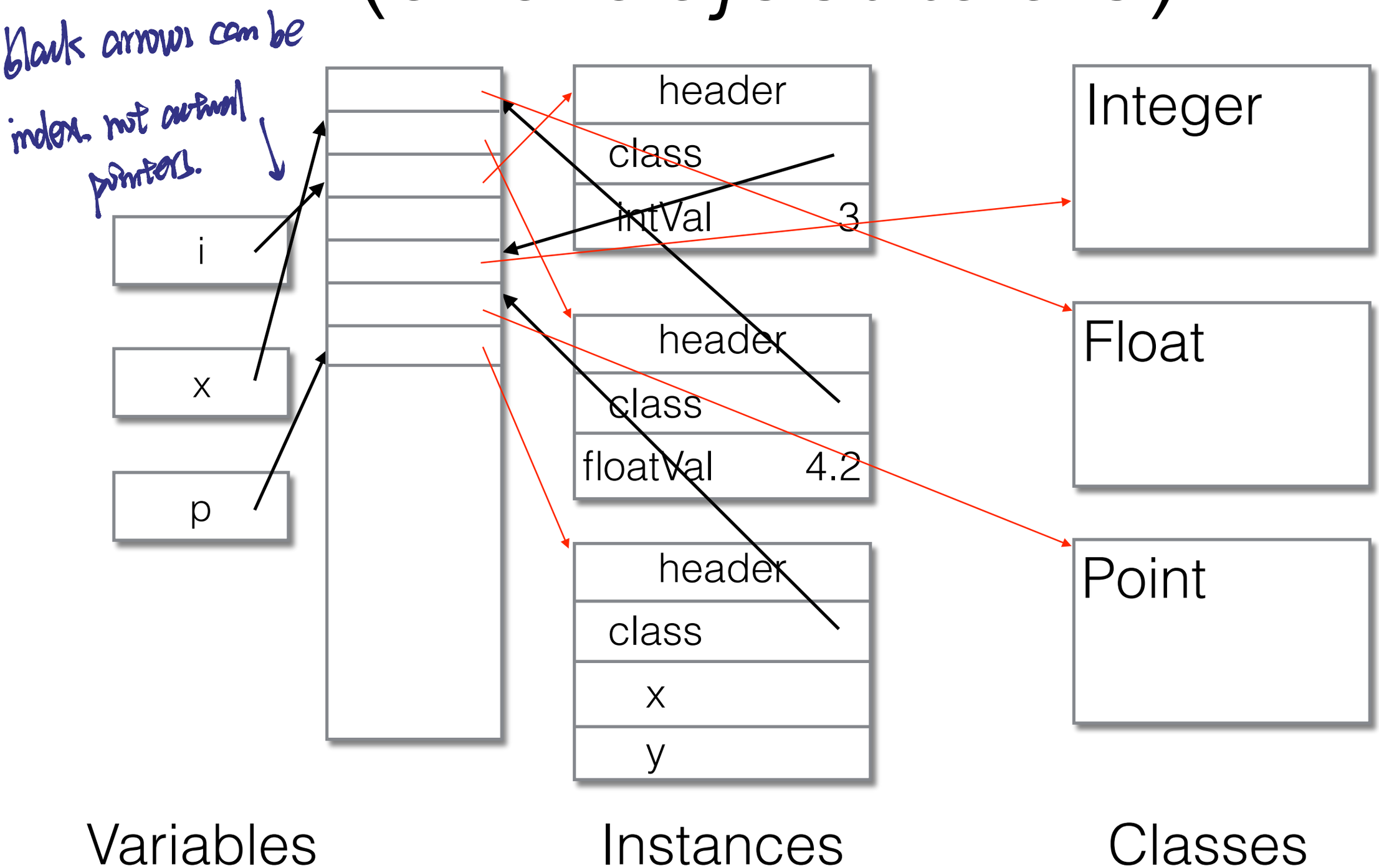
Indirect heap organization (aka *object table*)



Indirect heap organization (aka *object table*)



Indirect heap organization (aka *object table*)



big time penalty for memory.

Pros and cons of an object table

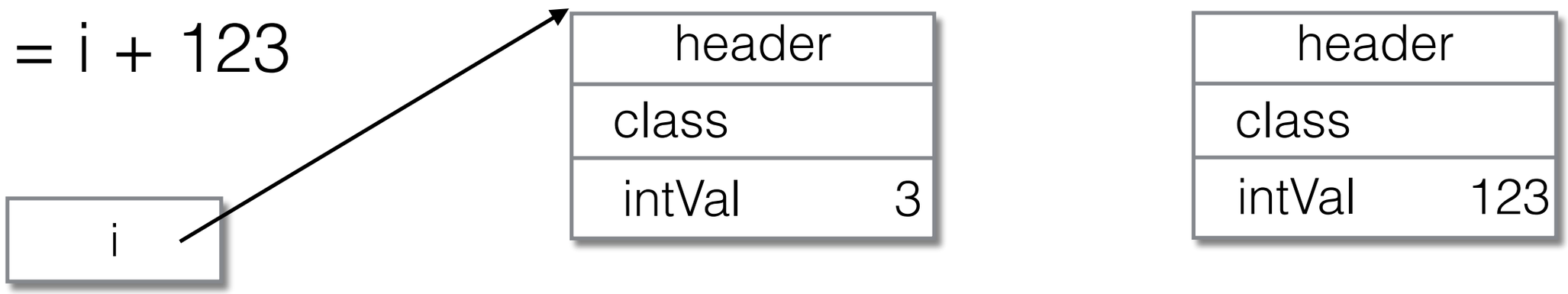
- Extra indirection: slow
- Extra memory: how much? avg: 7-10 fields.

+ Enables incremental relocation of objects (for GC, compaction)

Abandoned in high-speed systems following Ungar's 1986 UCB PhD thesis which established that the indirection cost is too high (and introduced a new GC technique using direct pointers).

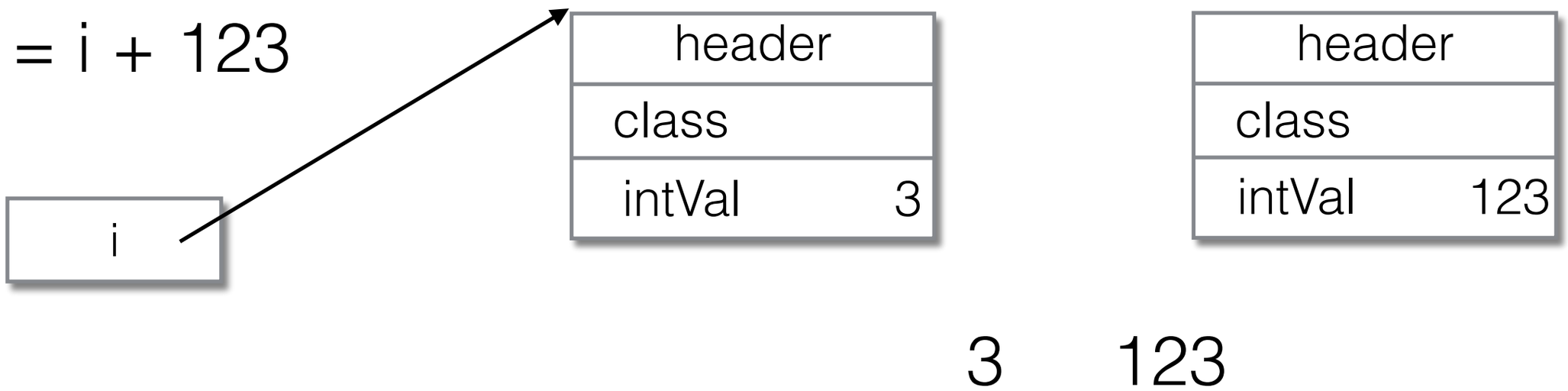
Example of boxed arithmetic

i = i + 123



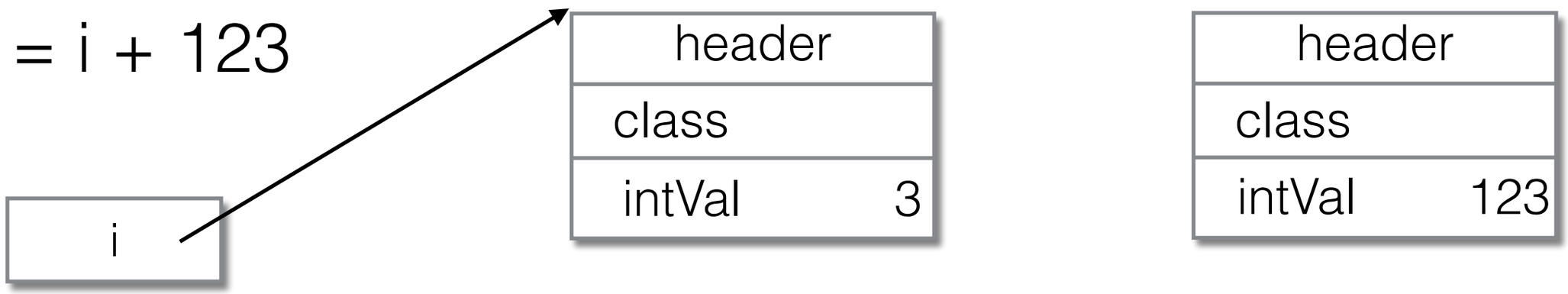
Example of boxed arithmetic

i = i + 123



Example of boxed arithmetic

i = i + 123



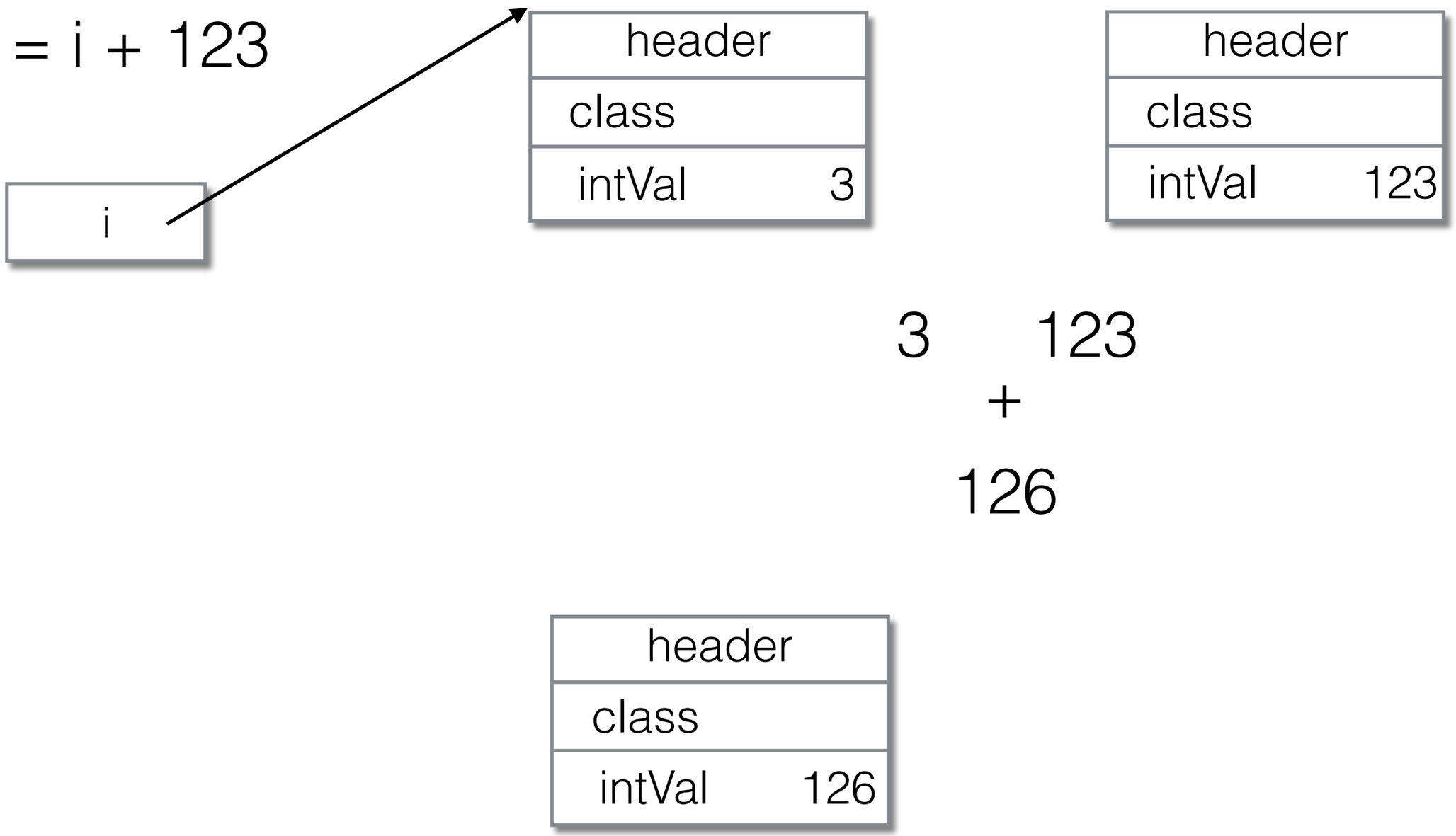
3 123

+

126

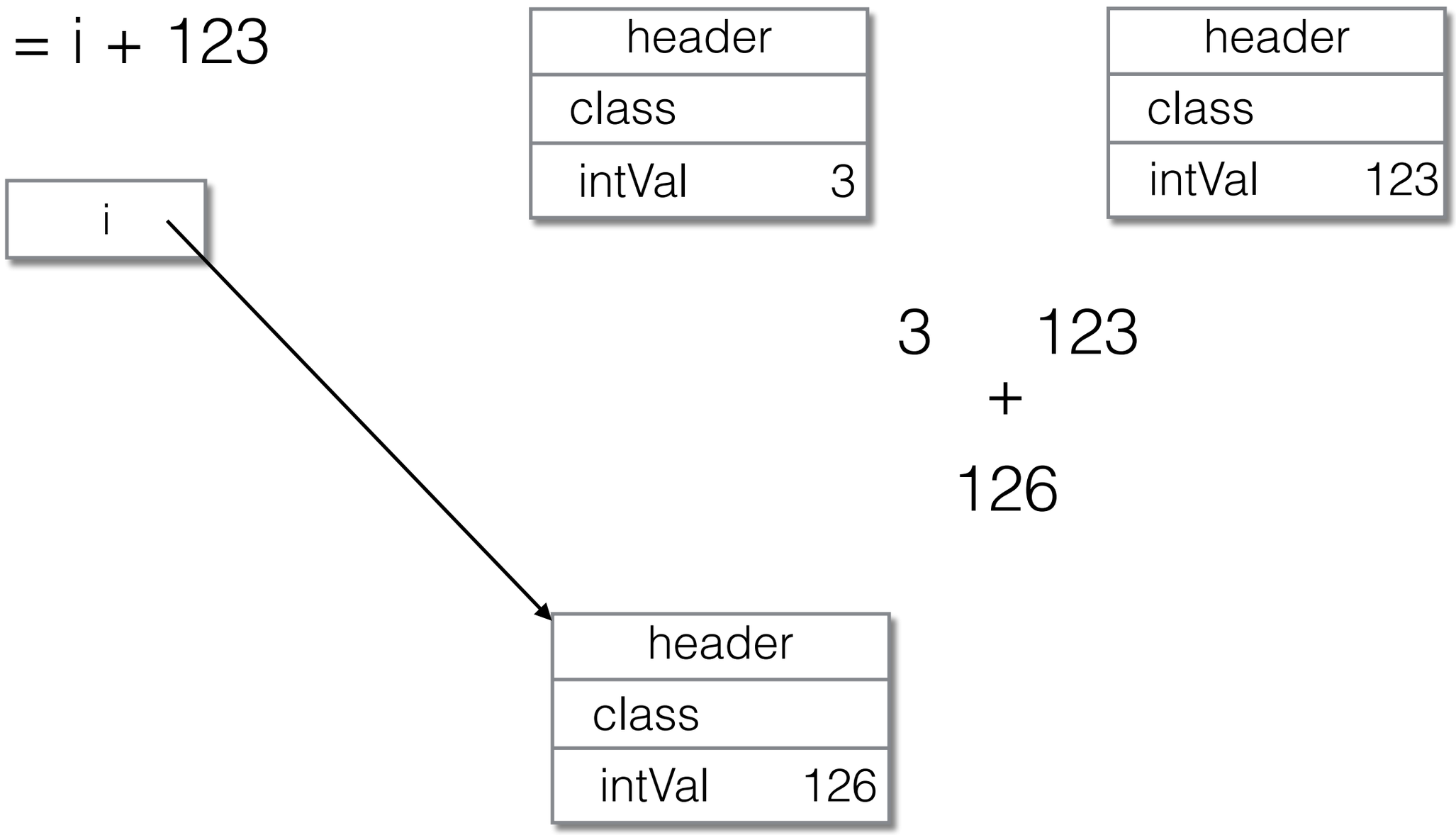
Example of boxed arithmetic

i = i + 123



Example of boxed arithmetic

i = i + 123



Primitive values

- Putting primitive values (integers, floats, chars) into the heap is very costly
  - Allocation and deallocation (GC) costs
  - Memory overhead (pointer and header and alignment)
  - Load and store penalty of indirection
- One alternative: Instead, restrict value range and use *tag bits* to distinguish pointers from primitive values

Tags

- Typically, addresses are word-aligned: e.g., on a 32-bit machine, objects start on 4-byte boundaries two bits on the bottom.
- Each address has two low-order zero bits
- Represent up to three types using the remaining three combinations of low-order bits, e.g.:
  - 00 - address
  - 01 - character
  - 10 - single-precision float
  - 11 - integer

this view is from the value of address, when encounter like

0x1000 → legal, address

while encounter like

0x8304242 ⇒ single precision-float.

Costs of tagging

- In this scheme, first have to test tag bits
  - object access can then proceed directly
  - integer arithmetic needs to strip tags
    right shift
  - floats - how to represent 32-bit floats in 30 bits?
    - reduce mantissa or exponent or both?

Alternate encodings

- An alternative scheme uses 00 for integers, 11 for addresses
  - Arithmetic doesn't need to strip or replace tags
  - Removing a tag before a load or store is a small overhead (and might not be necessary if using an indexed load to access a field)
- Inadvertently dereferencing a tagged address may result in a hardware exception due to the unaligned access.

Interpretation using tagged operands

```
typedef X word; // X should be an int type big enough
                // for all tagged primitive types

#define TAG 0x3
#define INT 0x0
#define FLOAT 0x1
#define OBJ 0x3

#define tag(w)        ((w)&TAG)
#define isInt(w)      (tag(w)==INT)
#define intVal(w)     ((int)((w)>>2))
#define setInt(i)     ((word)((i<<2)|INT))
#define isFloat(w)    (tag(w)==FLOAT)
float floatVal(word w); // extract float from float-tagged word
word setFloat(float f); // convert float to tagged word
#define isObj(w)      (tag(w)==OBJ)
#define obj(w)        ((word*)((w)&~TAG)
#define setObj(p)     ((((word)p)<<2)|INT)

word stack[MAXSTACK];
word vars[MAXVARS];
```

Expression bytecode interpreter

```
case o(LITINT): *++sp= setInt(get4int(pc+1));
                pc += 5;  continue;
case o(ADD):  if (isInt(sp[0]))
                if (isInt(sp[-1]))  // both ints
                  sp[-1]= setInt(intVal(sp[-1]) + intVal(sp[0]));
                else if (isFloat(sp[-1]))  // sp[-1] float, sp[0] int
                  sp[-1]= setFloat(floatVal(sp[-1]) + intVal(sp[0]));
                else
                  error("objects can't add");
              else if (isFloat(sp[0]))
                if (isFloat(sp[-1]))  // both floats
                  sp[-1]= setFloat(floatVal(sp[-1]) + floatVal(sp[0]);
                else if (isInt(sp[-1])  // sp[-1] int, sp[0] float
                  sp[-1]= setFloat(intVal(sp[-1]) + floatVal(sp[0]);
                else
                  error("objects can't add");
              else
                error("objects can't add");
              --sp; ++pc;
              continue;
```

Very straightforward.

Hardware support for tagging

- A variety of machines in the 1970s and 80s had hardware-supported tagging (Lisp Machines, esp.; also SOAR)

- SPARC included tag-checking operations (2 LSBs must be zero for ints):

  - `TADDcc`, `TSUBcc` Tagged Add/Subtract and modify icc

  - `TADDccTV`, `TSUBccTV` Tagged Add/Subtract, modify icc and Trap on Overflow

- Compilation techniques seem to have obviated these — see later.

eliminated

Exercise (to come)

- Modify your Feeny bytecode interpreter to use tagged arithmetic and measure the performance changes (time and space).

Absence of static type information: implications on **code**

- Don't know which target method(s) can be invoked at a call:
  - x.f() could invoke f() from *any* class or *be an error*
  - In general case, need to perform full lookup and dispatch
- Many operations are, under the covers, method calls (arithmetic, logic, perhaps even getters and setters and control flow) — the performance of this is *critical.*

Examples

- JavaScript
  `a+b` // float arithmetic; int as optimization; string concat

- Smalltalk
  `a+b` is a send of + to `a` with `b` as argument

- Self
  `max: a < b ifTrue: b False: a`
  control structures (if-then-else) are messages

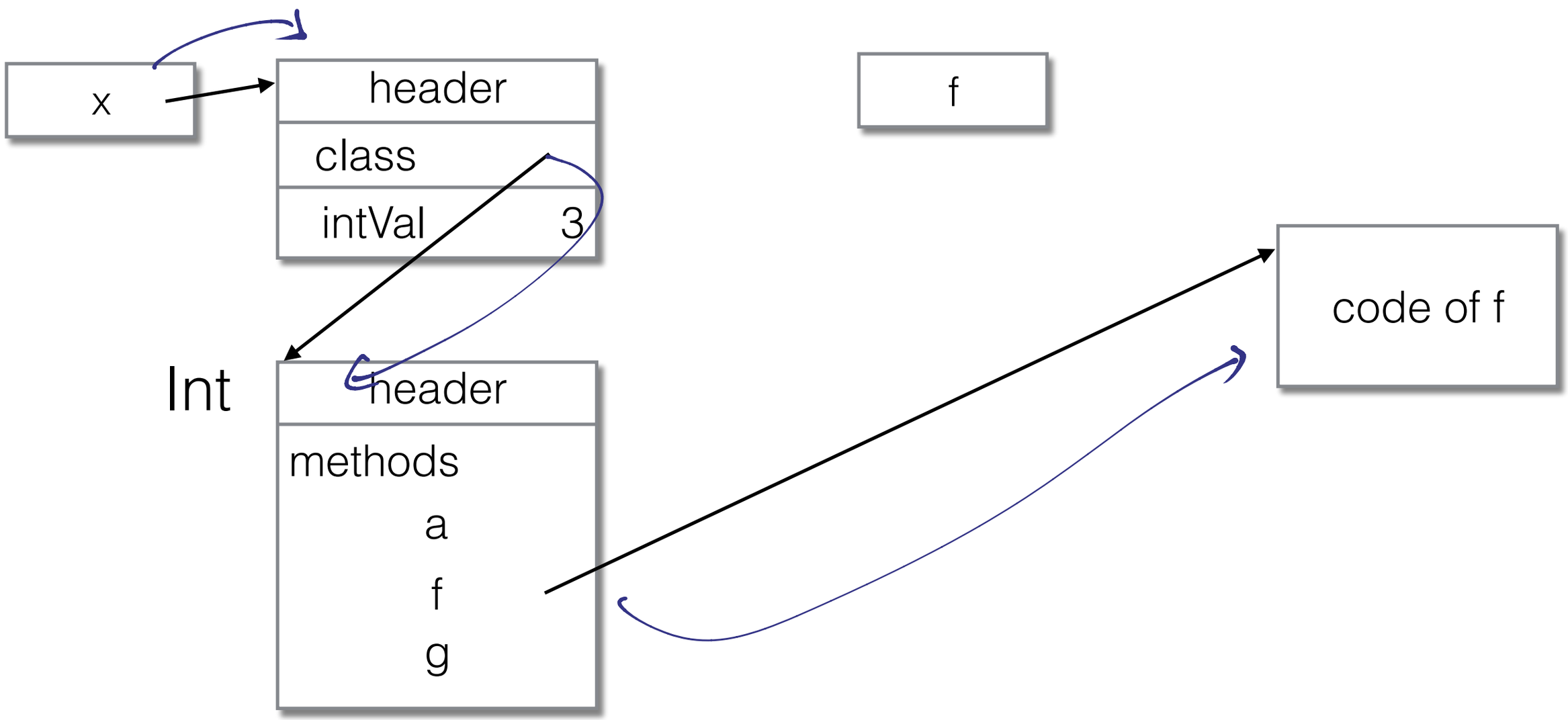
- R
  `a[i]` — indexing can invoke a method

Caching of call targets

- In the cases where the target method is determined by properties of the call (e.g., receiver's class and method name) we can use a *global method cache* for lookup

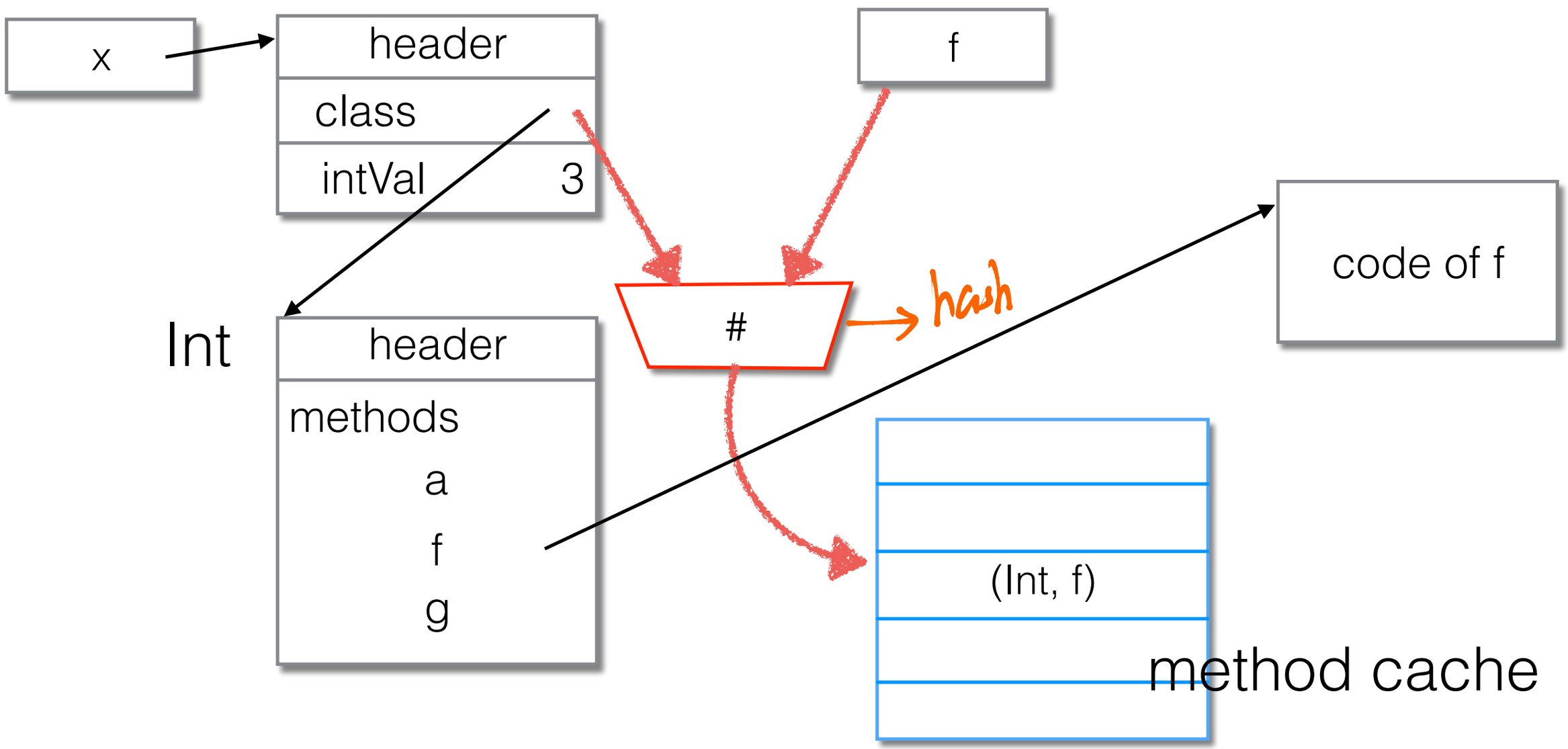
Global method cache

To invoke x.f, assuming f is defined by x's class:



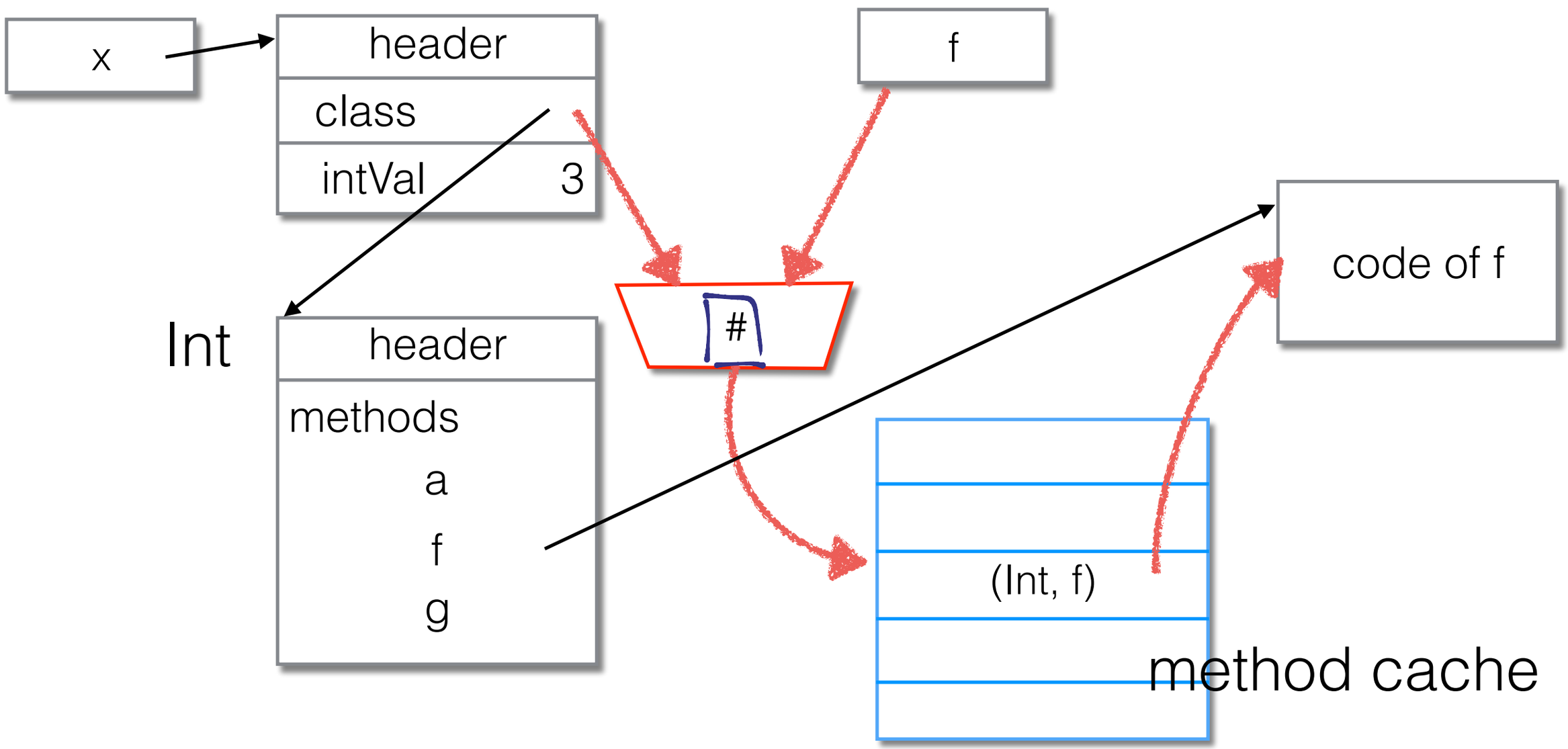
Global method cache

To invoke x.f, assuming f is defined by x's class:



Global method cache

To invoke x.f, assuming f is defined by x's class:



Example from a Smalltalk interpreter

```
static struct CacheEntry{
   OOP      Selector;
   OOP      Class;
   OOP      Method;
   BYTE     primIndex;
} MCache[CACHE_SIZE];
```

Smalltalk interpreter: send

```
void sndSelToClass(OOP Class) {
    WORD Hash= (msgSelector ^ Class) & (CACHE_SIZE - 1);
    struct CacheEntry *ce= &MCache[Hash];
    if (ce->Selector == msgSelector && ce->Class == Class) {
        // hit
        newMethod= ce->Method;
        primIndex= ce->primIndex;
    } else { // miss, do full lookup
        lookupMethodInClass(Class); // sets newMethod,primIndex
        ce->Selector  = msgSelector;
        ce->Class     = Class;
        ce->Method    = newMethod;
        ce->primIndex = primIndex;
    }
    executeNewMethod();
}
```

Caching of call targets — the call-site cache

- An alternative scheme is a call-site cache, which was introduced in the Deutsch-Schiffman paper (assigned reading).
- We will look at this, and other techniques, when we study dynamic compilation.

Exercise

- Add a method lookup cache to your Feeny bytecode interpreter and measure the change in speed. How big should the cache be?

Reading assignment

- Read *Efficient Implementation of the Smalltalk-80 System* by L Peter Deutsch and Allan Schiffman, http://dx.doi.org/10.1145/800017.800542.
  Be prepared to answer questions if called upon.
  Optional background reading is in the bibliography.

  Next week the authors will join us for a discussion of the paper. Post questions for them (questions not addressed in the paper) to Piazza (in response to my posting) by the end of Thursday; I will moderate the discussion and include relevant questions.