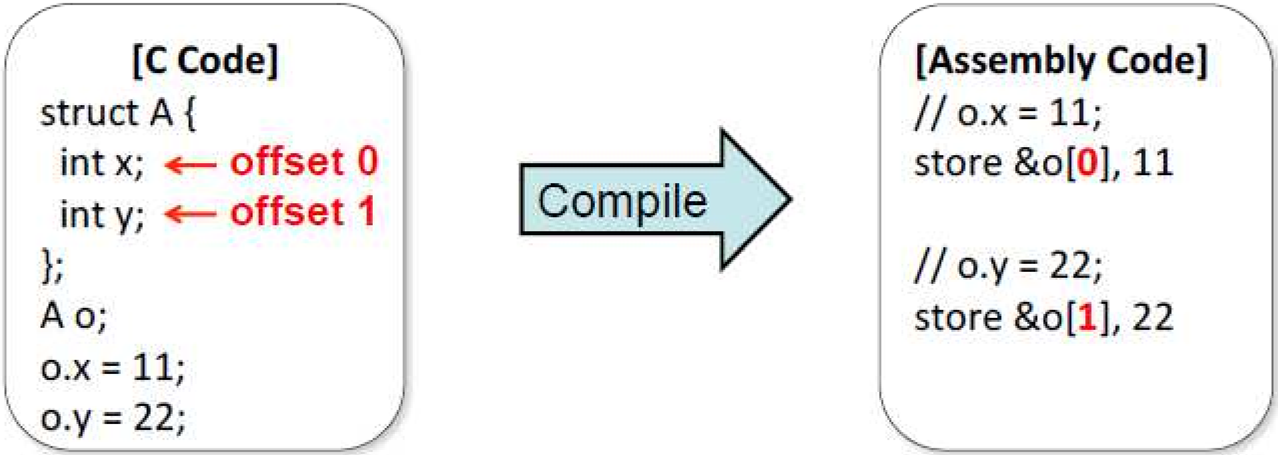
Optimizations

Why traditional languages are fast

- Types tell compiler the shape of *o (fields and their offsets)*



[C Code]

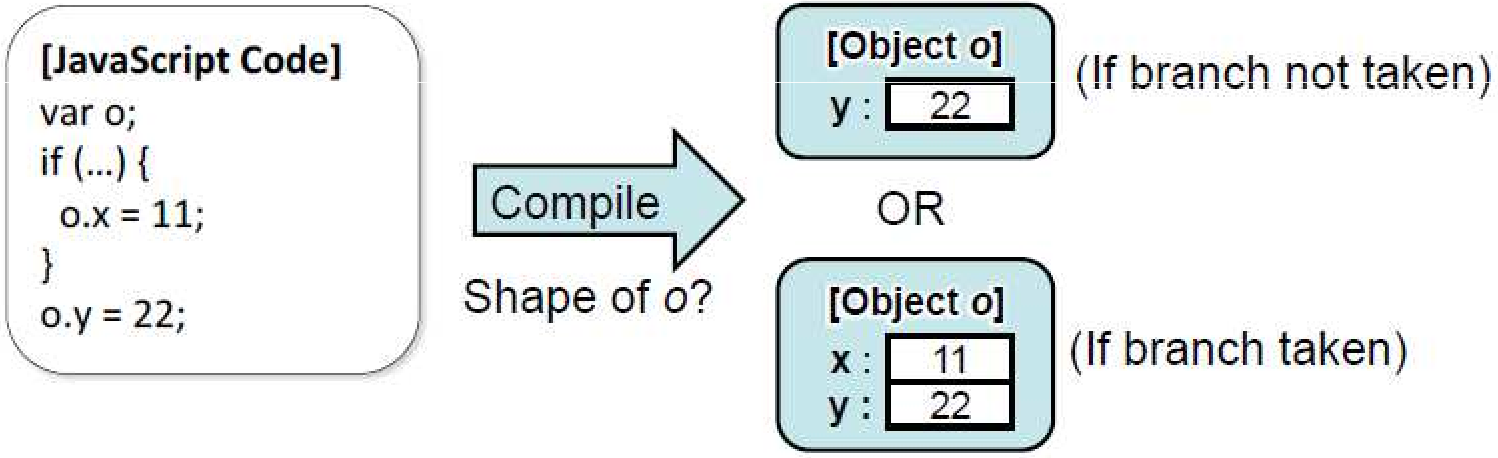
```
struct A {
  int x;  ← offset 0
  int y;  ← offset 1
};
A o;
o.x = 11;
o.y = 22;
```

[Assembly Code]

```
// o.x = 11;
store &o[0], 11

// o.y = 22;
store &o[1], 22
```

Scripting Languages Have No Types

- Objects are simply dictionaries from properties to values
- Properties can be added and removed at any time

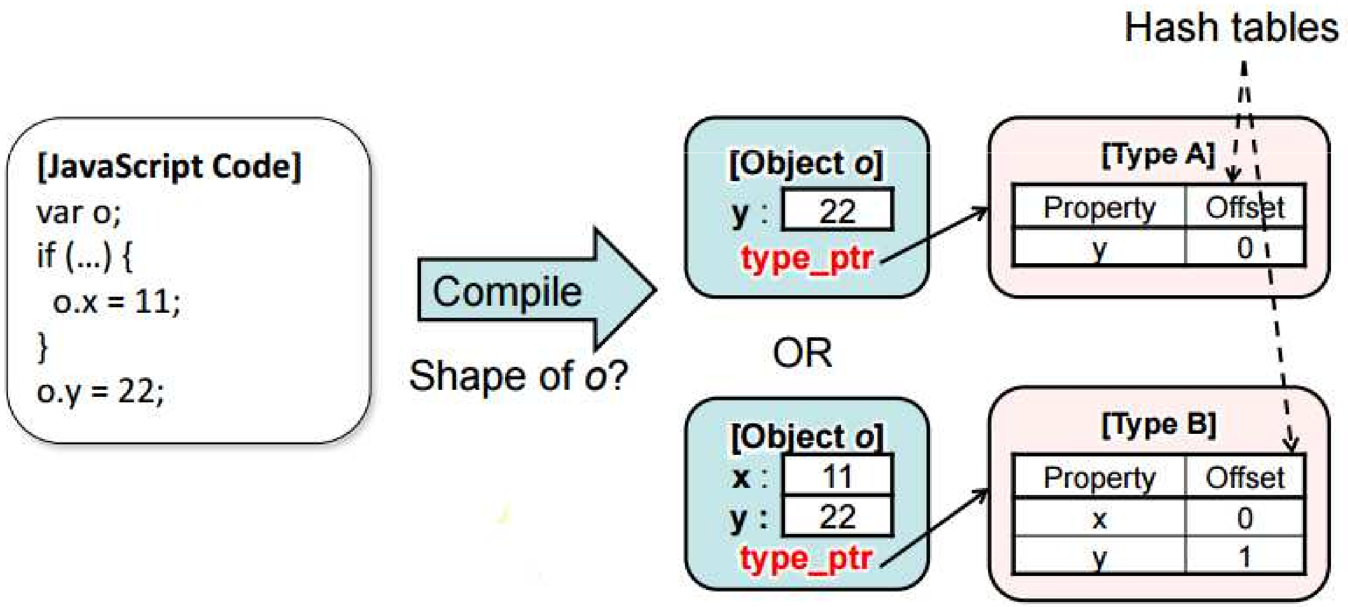


```
[JavaScript Code]
var o;
if (...) {
  o.x = 11;
}
o.y = 22;
```

- How to generate code when shape of o is unknown?

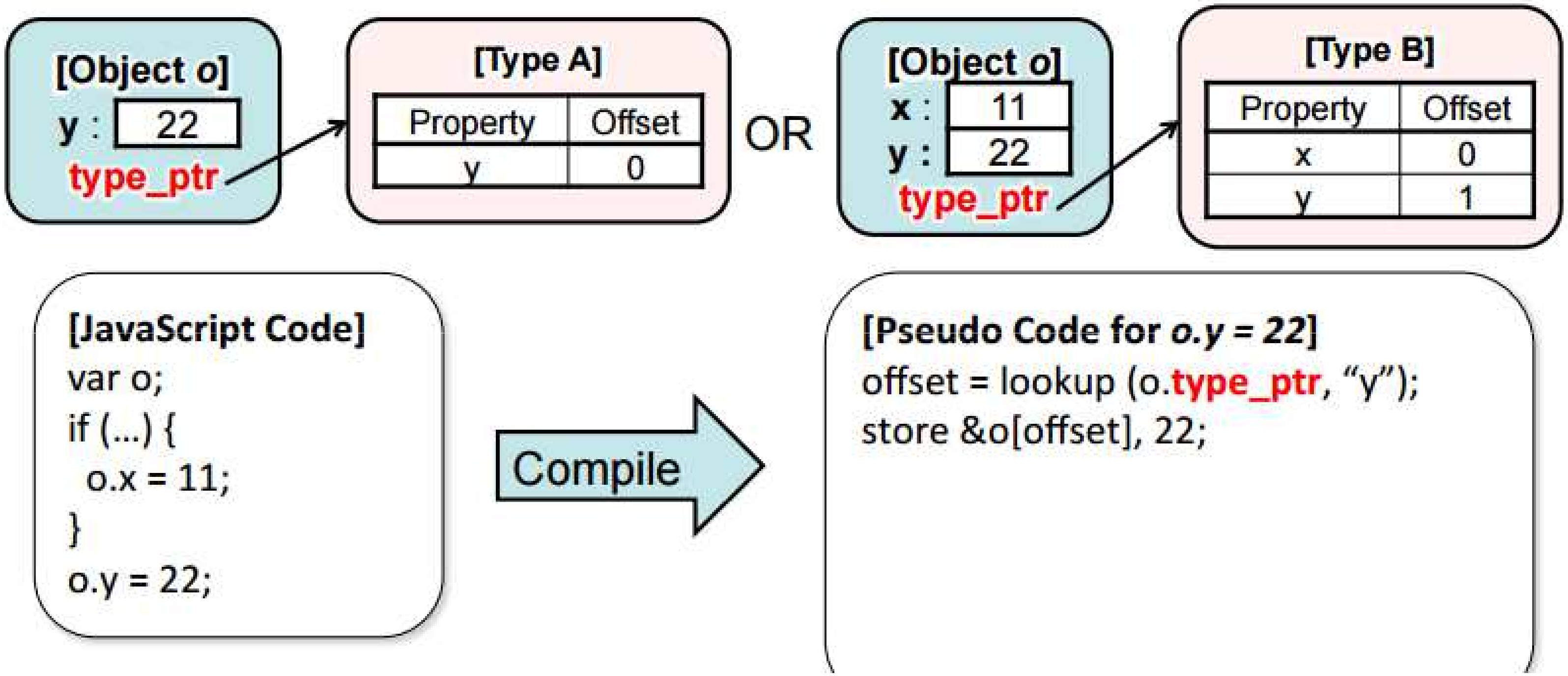
Scripting Languages Have No Types

- Objects are simply dictionaries from properties to values
- Properties can be added and removed at any time



- Compilers introduce a type system behind the scenes

Scripting Languages Have No Types -> Slow



[Object o]
y : 22
type_ptr

[Type A]

Property	Offset
y	0

OR

[Object o]
x : 11
y : 22
type_ptr

[Type B]

Property	Offset
x	0
y	1

[JavaScript Code]

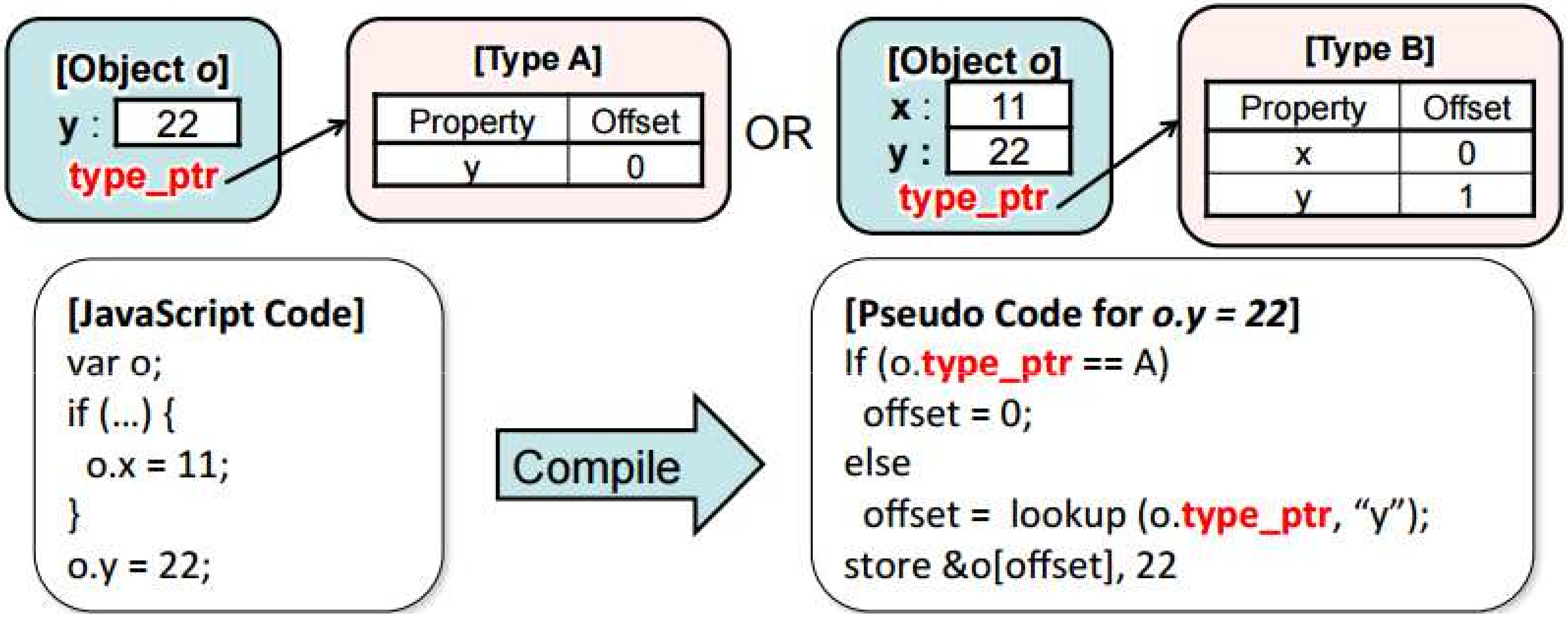
```
var o;
if (…) {
  o.x = 11;
}
o.y = 22;
```

[Pseudo Code for *o.y = 22*]

```
offset = lookup (o.type_ptr, “y”);
store &o[offset], 22;
```

- A field access always entails a hash table lookup to get the offset

State-of-the-art Compilers do Type Specialization



[Object o]
y : 22
type_ptr

[Type A]

Property	Offset
y	0

OR

[Object o]
x : 11
y : 22
type_ptr

[Type B]

Property	Offset
x	0
y	1

[JavaScript Code]

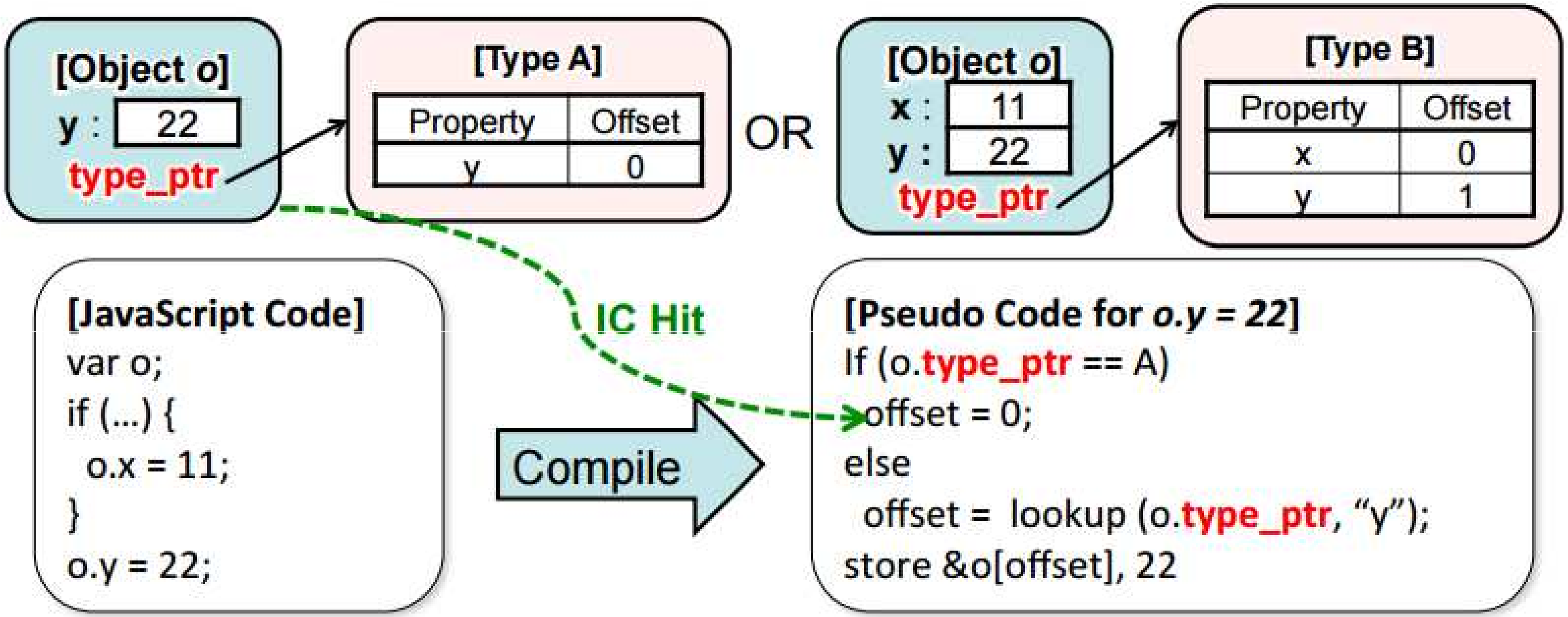
```
var o;
if (…) {
  o.x = 11;
}
o.y = 22;
```

[Pseudo Code for *o.y = 22*]

```
If (o.type_ptr == A)
  offset = 0;
else
  offset =  lookup (o.type_ptr, "y");
store &o[offset], 22
```

- Type specialization: Optimizing code to be fast for the recorded types
- Inline Cache (IC): Actual optimized code

State-of-the-art Compilers do Type Specialization



[Object o]
y : 22
type_ptr

[Type A]

Property	Offset
y	0

OR

[Object o]
x : 11
y : 22
type_ptr

[Type B]

Property	Offset
x	0
y	1

[JavaScript Code]

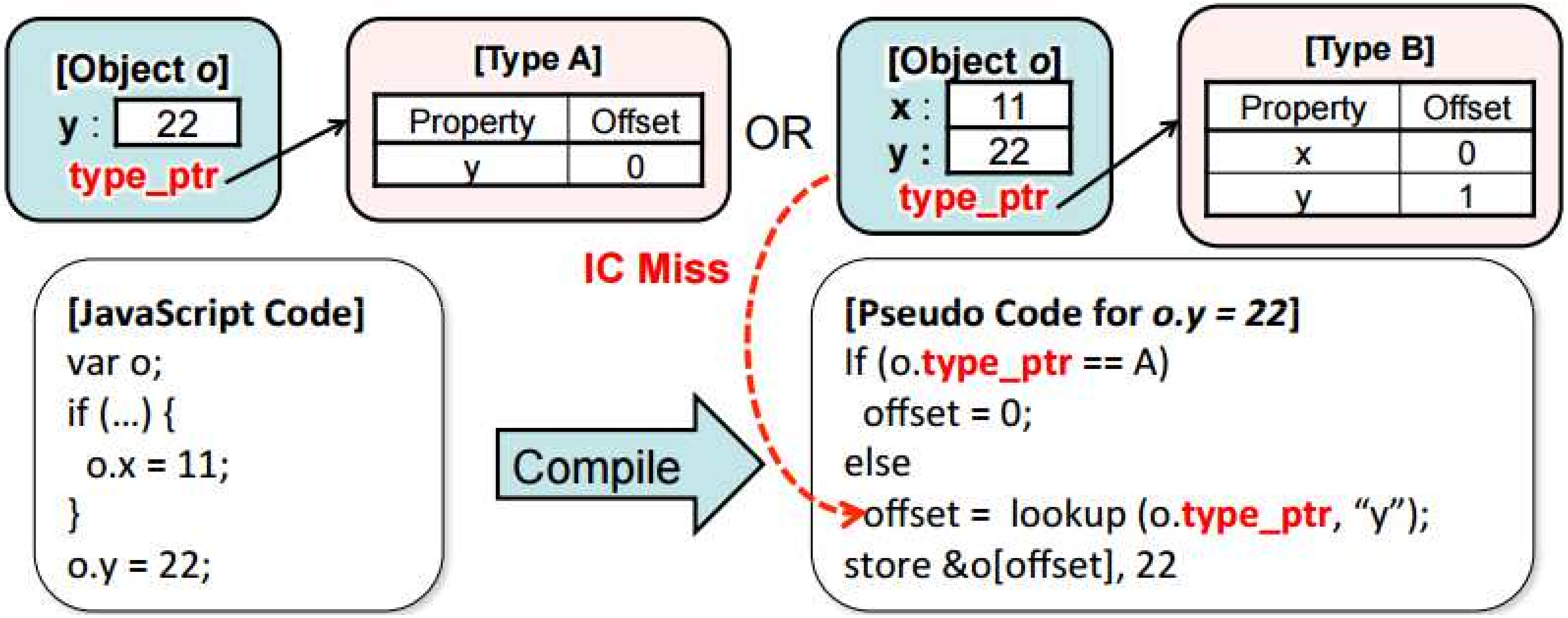
```
var o;
if (…) {
  o.x = 11;
}
o.y = 22;
```

[Pseudo Code for *o.y = 22*]

```
If (o.type_ptr == A)
  offset = 0;
else
  offset =  lookup (o.type_ptr, "y");
store &o[offset], 22
```

- IC Hit: Execution encounters one of the recorded types -> Fast

State-of-the-art Compilers do Type Specialization



[Type A]

Property	Offset
y	0

[Type B]

Property	Offset
x	0
y	1

[JavaScript Code]

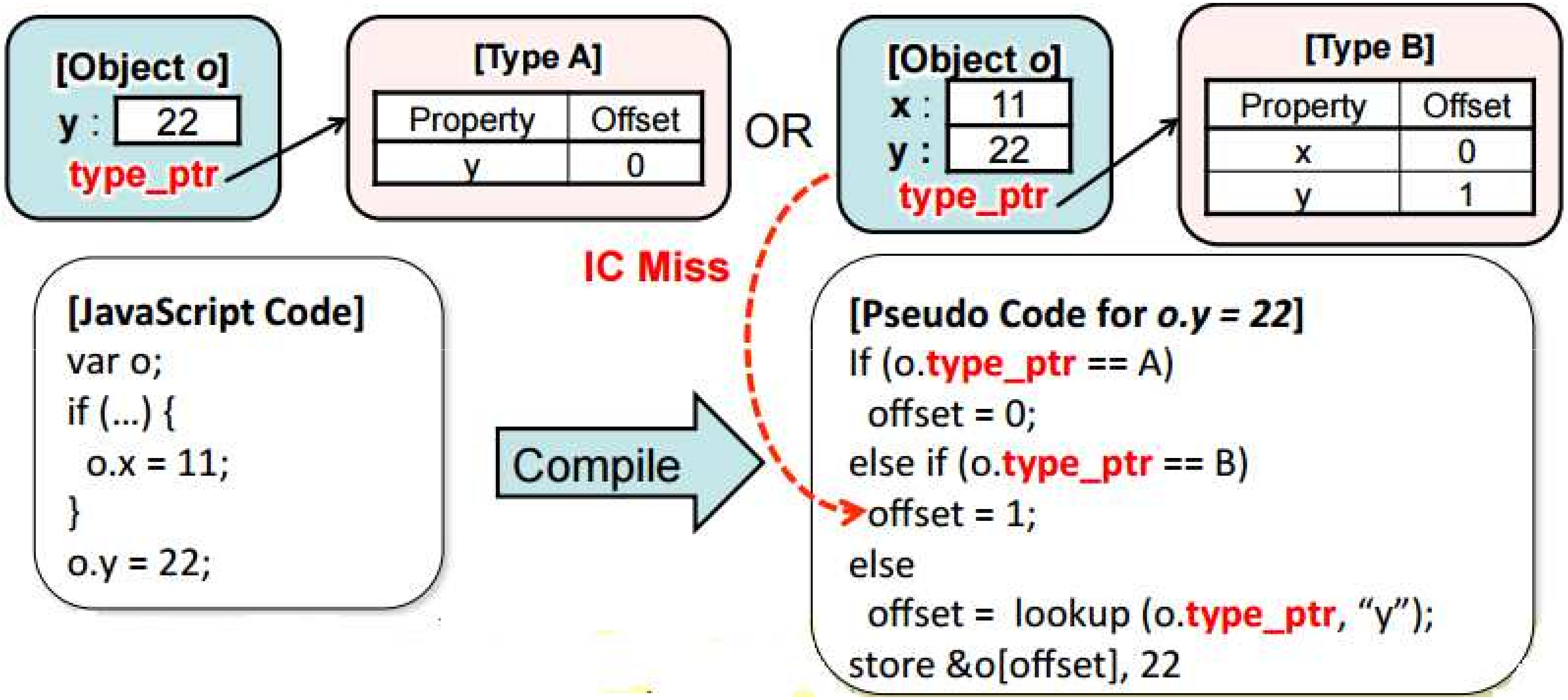
```
var o;
if (…) {
  o.x = 11;
}
o.y = 22;
```

[Pseudo Code for *o.y = 22*]

```
If (o.type_ptr == A)
  offset = 0;
else
  offset =  lookup (o.type_ptr, “y”);
store &o[offset], 22
```

- IC Miss: Execution results in a hash table lookup -> Very Slow

State-of-the-art Compilers do Type Specialization



[Type A]

Property	Offset
y	0

[Type B]

Property	Offset
x	0
y	1

[JavaScript Code]

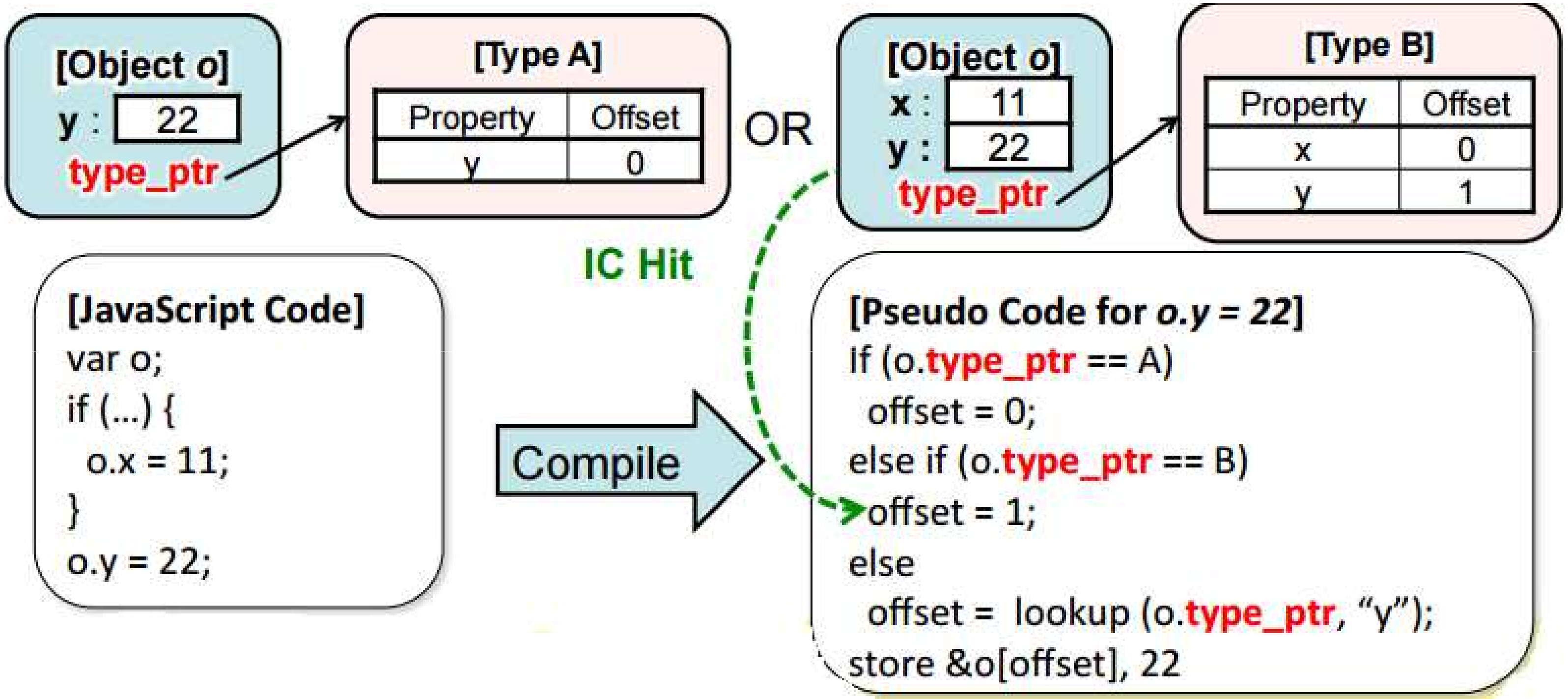
```
var o;
if (…) {
  o.x = 11;
}
o.y = 22;
```

[Pseudo Code for *o.y = 22*]

```
If (o.type_ptr == A)
  offset = 0;
else if (o.type_ptr == B)
  offset = 1;
else
  offset =  lookup (o.type_ptr, "y");
store &o[offset], 22
```

- IC Miss: Execution results in a hash table lookup -> Very Slow

State-of-the-art Compilers do Type Specialization



[Object o]
y : 22
type_ptr

[Type A]

Property	Offset
y	0

OR

[Object o]
x : 11
y : 22
type_ptr

[Type B]

Property	Offset
x	0
y	1

[JavaScript Code]

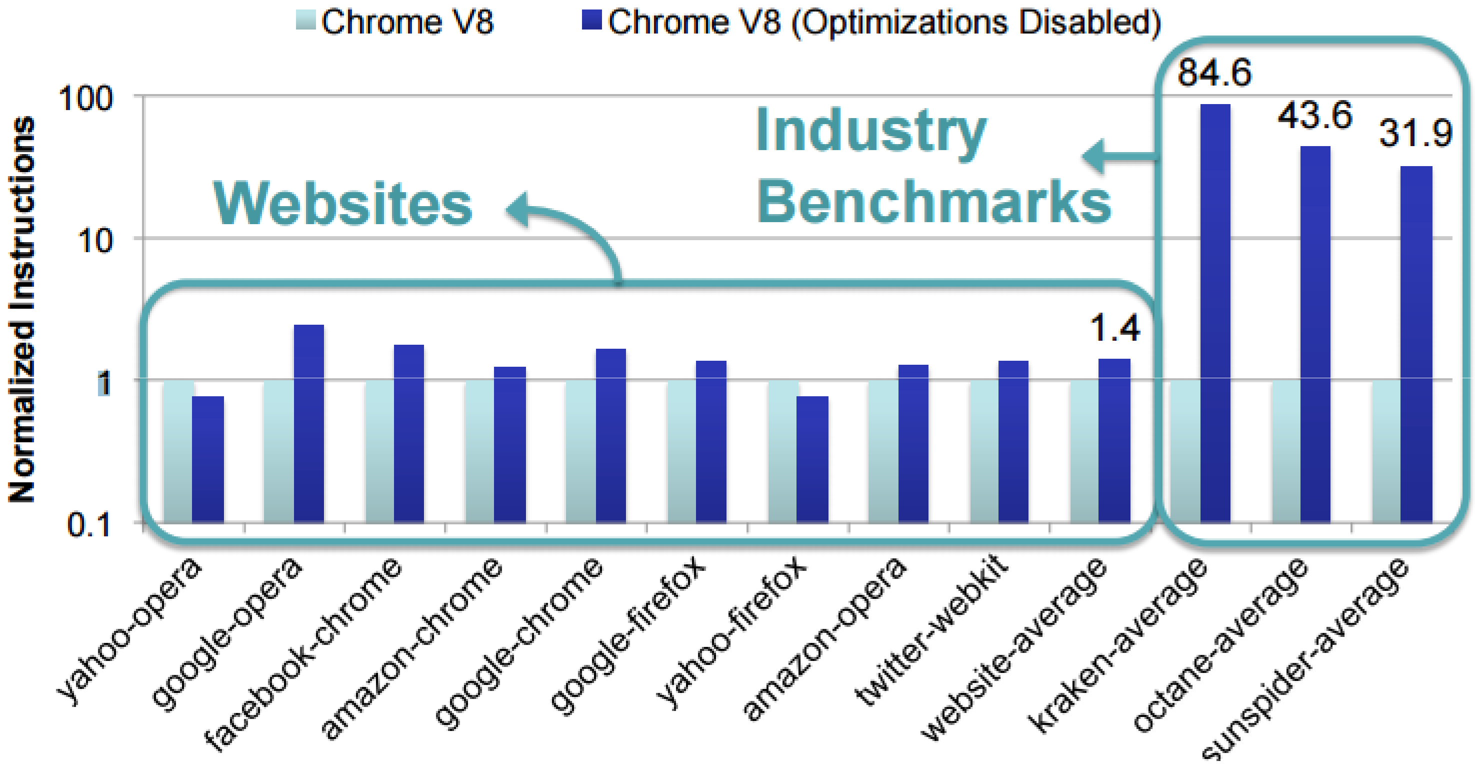
```
var o;
if (…) {
  o.x = 11;
}
o.y = 22;
```

[Pseudo Code for *o.y = 22*]

```
If (o.type_ptr == A)
  offset = 0;
else if (o.type_ptr == B)
  offset = 1;
else
  offset =  lookup (o.type_ptr, “y”);
store &o[offset], 22
```

- IC Miss: Execution results in a hash table lookup -> Very Slow

Google Chrome V8: Ineffective for Many Real Websites

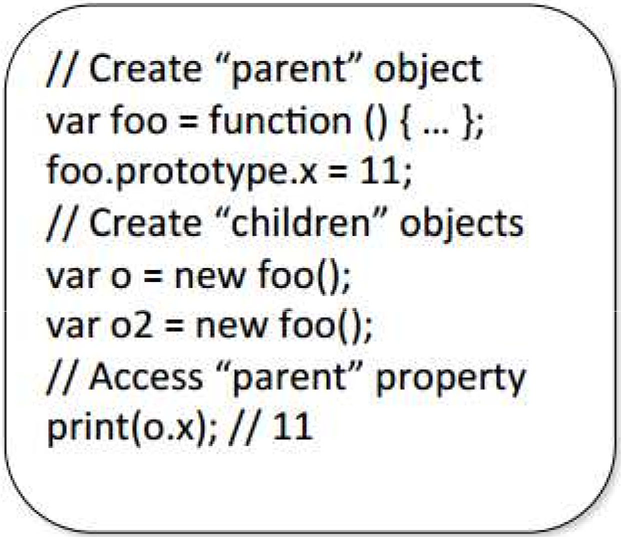


- Chrome V8 compiler not optimized for dynamism in real websites

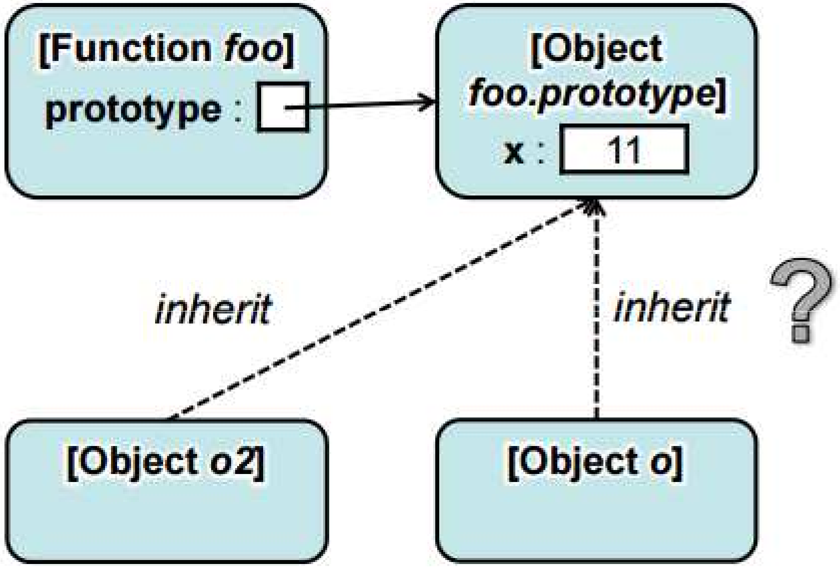
Chrome V8 Type System is Too Brittle

- Chrome V8 type system encodes (other than properties):
  - Inheritance (i.e. address of parent object)
  - Method bindings (i.e. addresses of functions called)
- Helps in generating efficient code during type specialization
  - Inheritance: helps when accessing parent properties
  - Method bindings: helps resolve targets for method calls
- V8 Assumption: inheritance and method bindings rarely change
  - Reasonable since always true for statically typed languages

Inheritance in JavaScript: Prototype Objects

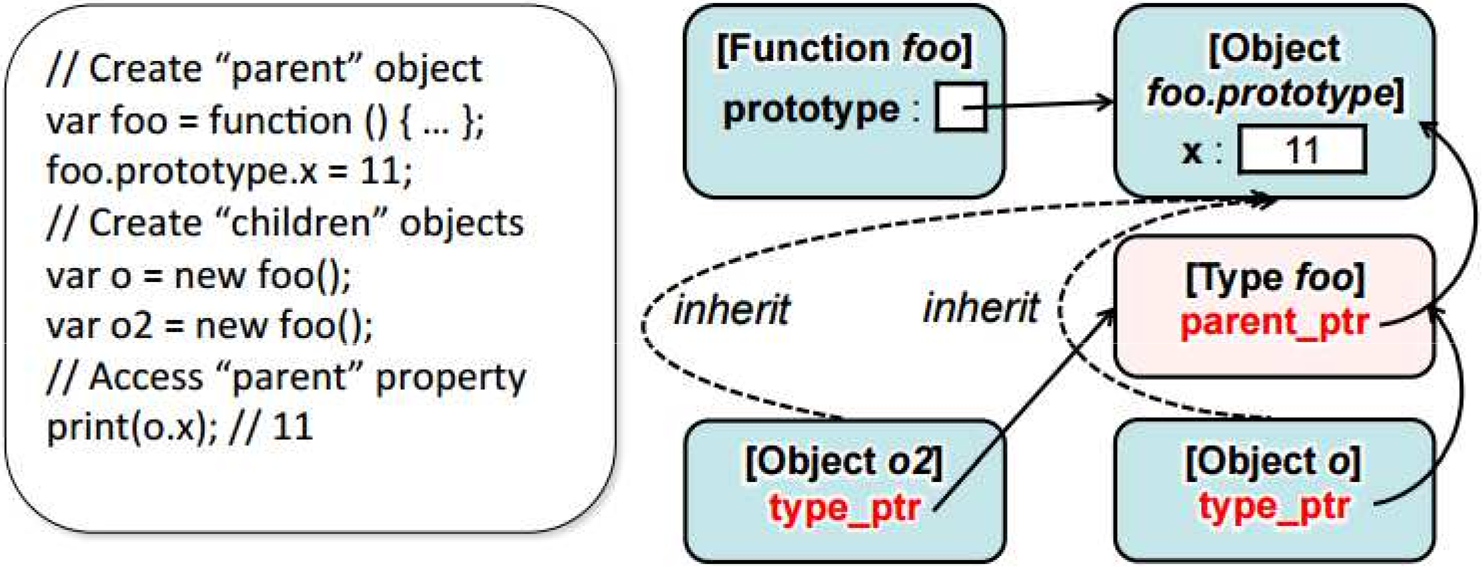


```
// Create "parent" object
var foo = function () { ... };
foo.prototype.x = 11;
// Create "children" objects
var o = new foo();
var o2 = new foo();
// Access "parent" property
print(o.x); // 11
```



1. Create function foo (implicitly creates "parent" foo.prototype)
2. Create "child" o by calling foo (o inherits from foo.prototype)
3. Access "parent" property x in foo.prototype through inheritance

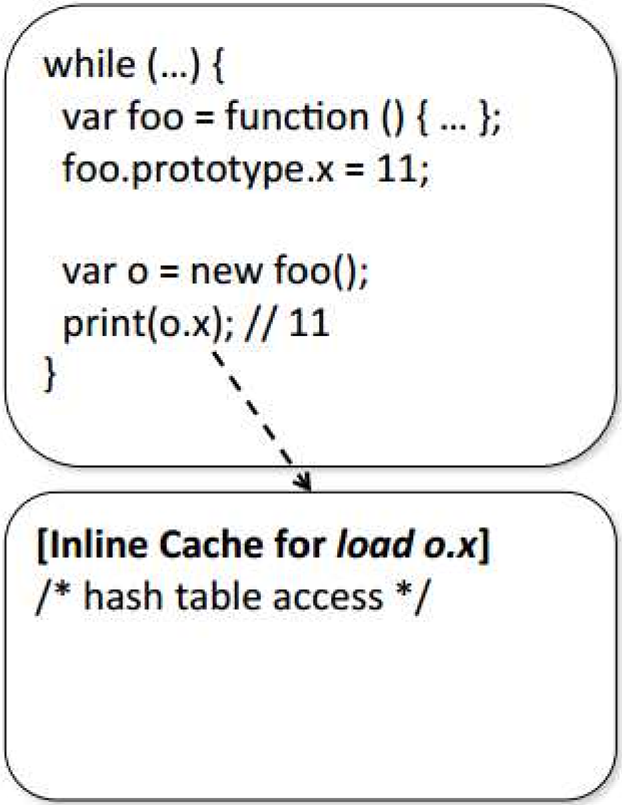
How V8 Encodes Inheritance into Types



```
// Create "parent" object
var foo = function () { ... };
foo.prototype.x = 11;
// Create "children" objects
var o = new foo();
var o2 = new foo();
// Access "parent" property
print(o.x); // 11
```

1. Create type that inherits foo.prototype (by linking through parent_ptr)
2. Have all objects created by foo() have that type
3. Access parent properties through type_ptr and parent_ptr links

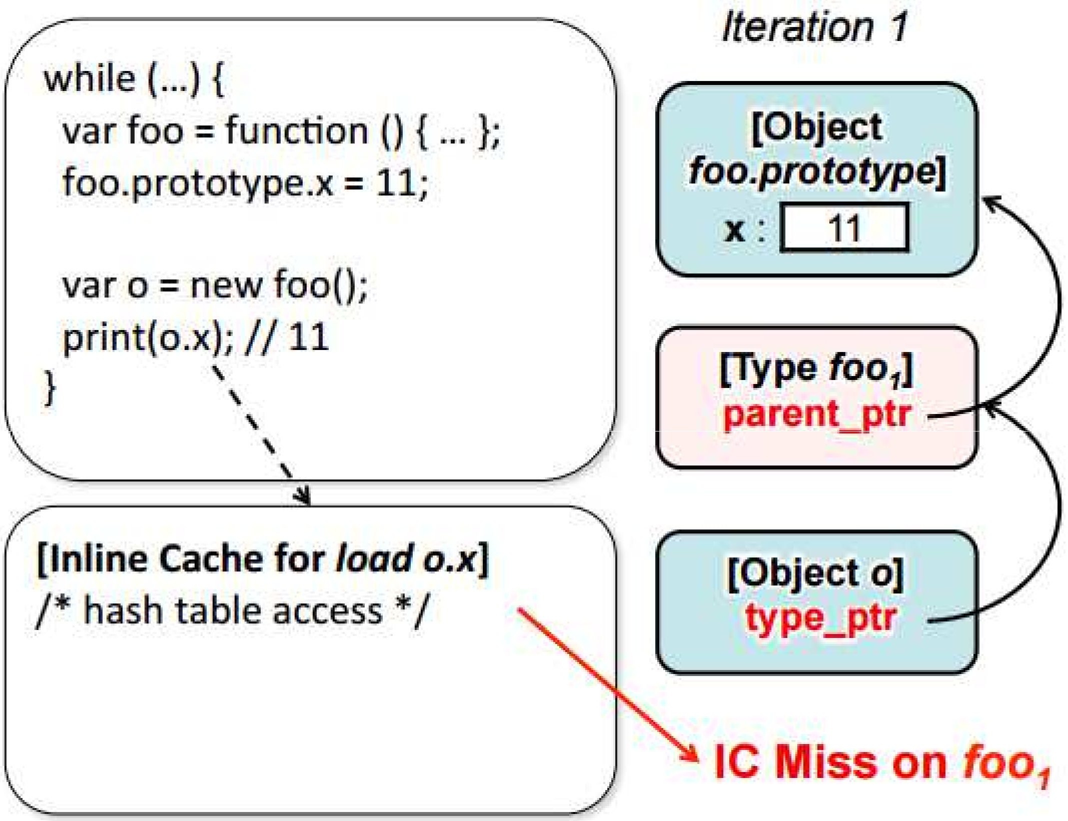
Problem: Rampant Type Creation



```
while (…) {
  var foo = function () { … };
  foo.prototype.x = 11;

  var o = new foo();
  print(o.x); // 11
}
```

[Inline Cache for *load o.x*]
/* hash table access */

Problem: Rampant Type Creation



```
while (...) {
  var foo = function () { ... };
  foo.prototype.x = 11;

  var o = new foo();
  print(o.x); // 11
}
```

[Inline Cache for *load o.x*]
/* hash table access */

Iteration 1

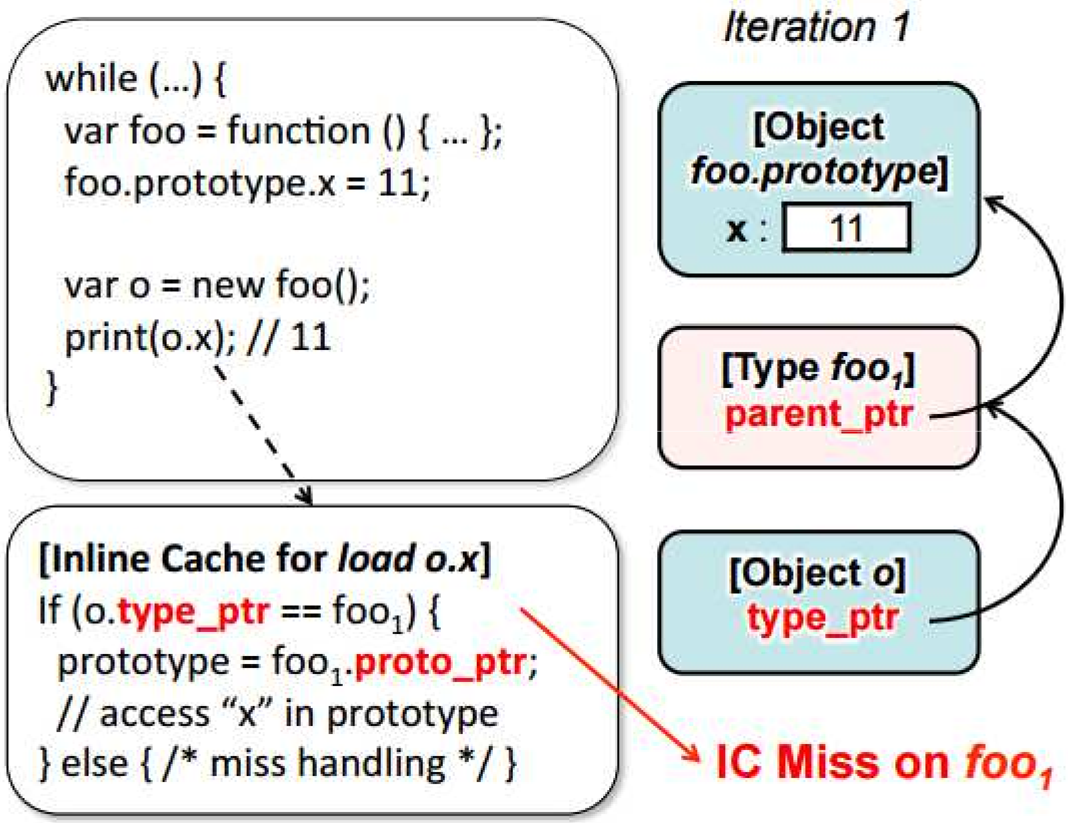
[Object *foo.prototype*]
x : 11

[Type *foo₁*]
parent_ptr

[Object *o*]
type_ptr

IC Miss on *foo₁*

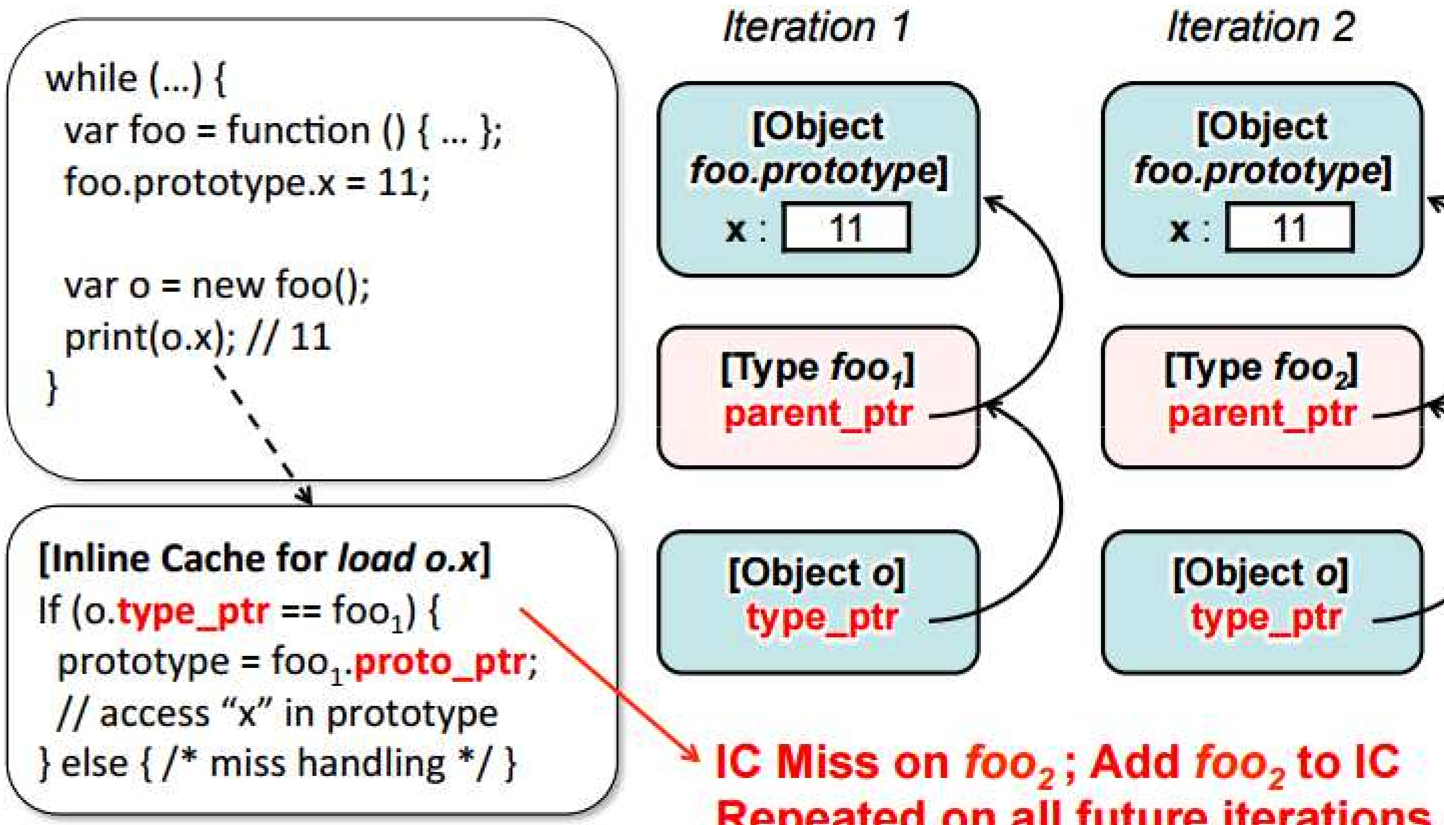
Problem: Rampant Type Creation



```
while (…) {
  var foo = function () { … };
  foo.prototype.x = 11;

  var o = new foo();
  print(o.x); // 11
}
```

```
[Inline Cache for load o.x]
If (o.type_ptr == foo1) {
  prototype = foo1.proto_ptr;
  // access “x” in prototype
} else { /* miss handling */ }
```

Problem: Rampant Type Creation



```
while (...) {
  var foo = function () { ... };
  foo.prototype.x = 11;

  var o = new foo();
  print(o.x); // 11
}
```

[Inline Cache for *load o.x*]

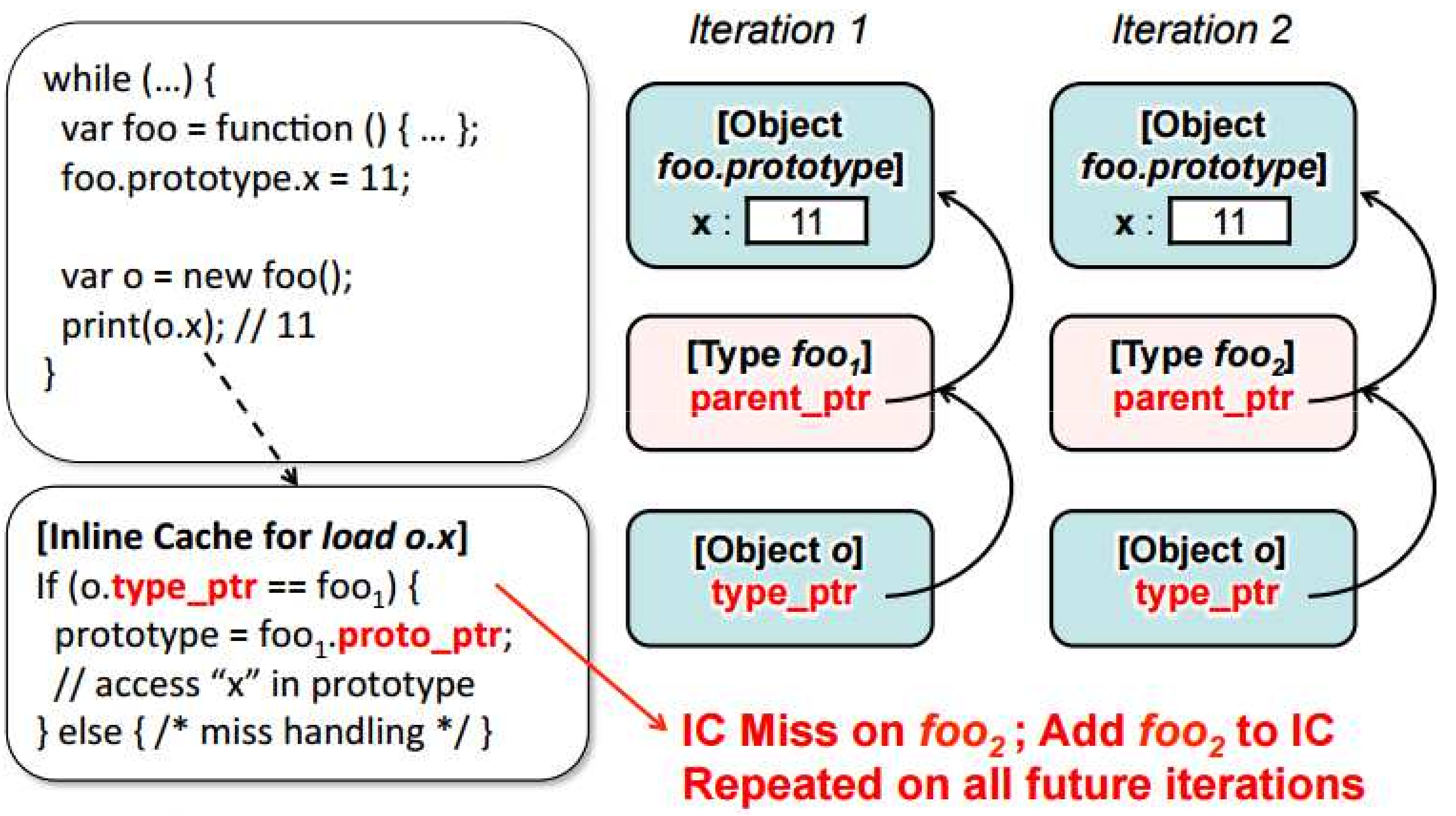
```
If (o.type_ptr == foo1) {
  prototype = foo1.proto_ptr;
  // access "x" in prototype
} else { /* miss handling */ }
```

IC Miss on *foo*$_2$; Add *foo*$_2$ to IC
Repeated on all future iterations

Problem: Rampant Type Creation

1. Very counterintuitive: same code creates different types
2. Dynamic function creation: common pattern in website code
   - For encapsulation and ease of programming

Problem: Rampant Type Creation



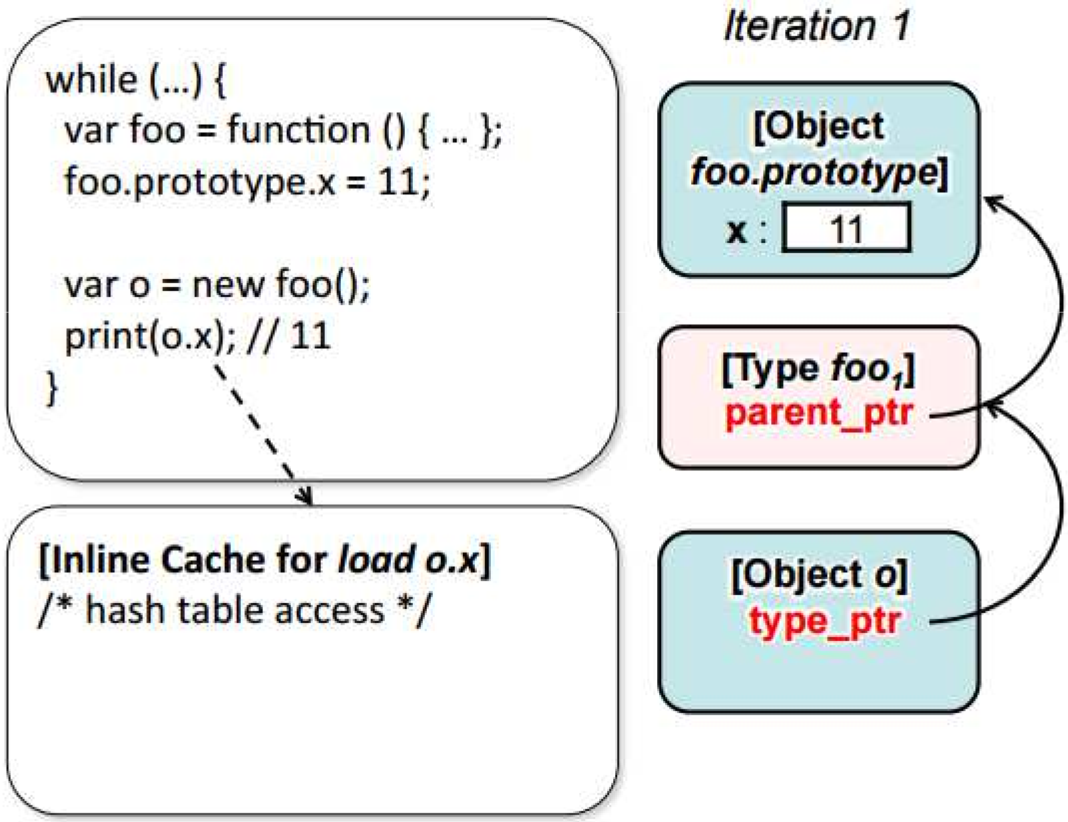
```
while (…) {
  var foo = function () { … };
  foo.prototype.x = 11;

  var o = new foo();
  print(o.x); // 11
}
```

[Inline Cache for *load o.x*]

```
If (o.type_ptr == foo_1) {
  prototype = foo_1.proto_ptr;
  // access "x" in prototype
} else { /* miss handling */ }
```

IC Miss on foo_2; Add foo_2 to IC
Repeated on all future iterations

Solution: Decouple Inheritance from Types



```
while (...) {
  var foo = function () { ... };
  foo.prototype.x = 11;

  var o = new foo();
  print(o.x); // 11
}
```

[Inline Cache for *load o.x*]
/* hash table access */

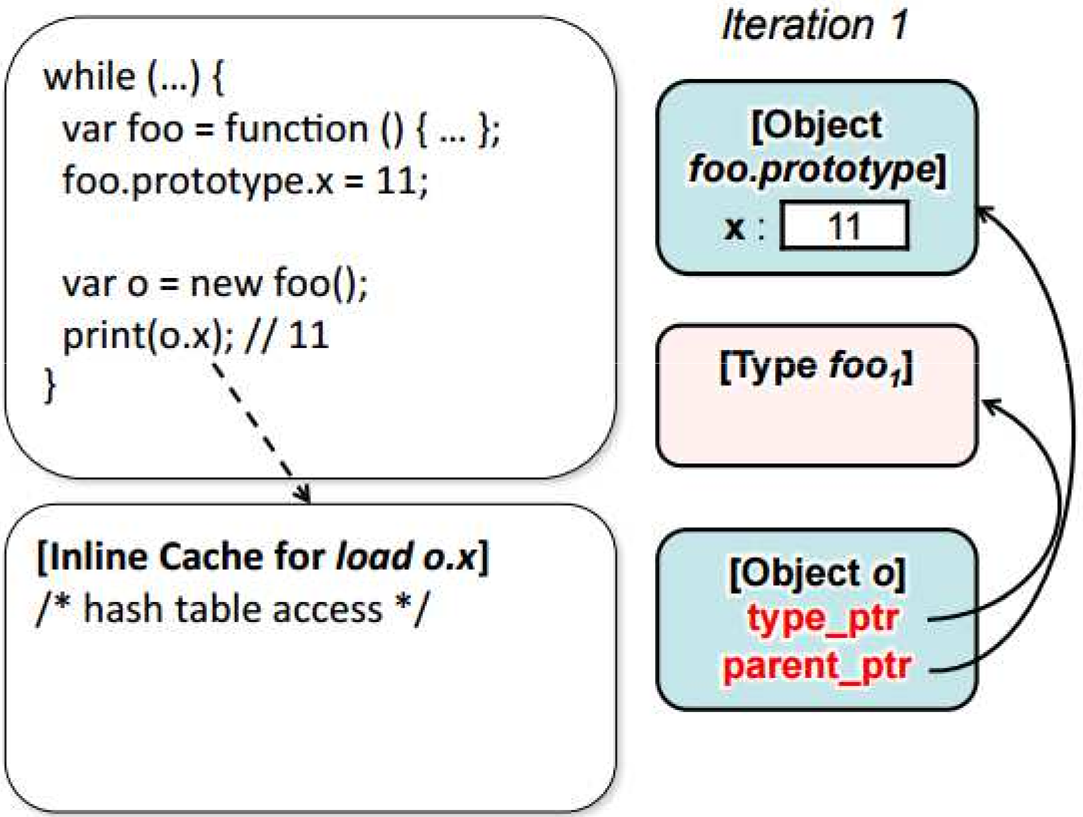
Iteration 1

[Object *foo.prototype*]
x : 11

[Type *foo*$_1$]
parent_ptr

[Object *o*]
type_ptr

Solution: Decouple Inheritance from Types



Iteration 1

```
while (...) {
  var foo = function () { ... };
  foo.prototype.x = 11;

  var o = new foo();
  print(o.x); // 11
}
```

[Inline Cache for *load o.x*]

/* hash table access */

[Object *foo.prototype*]

x : 11

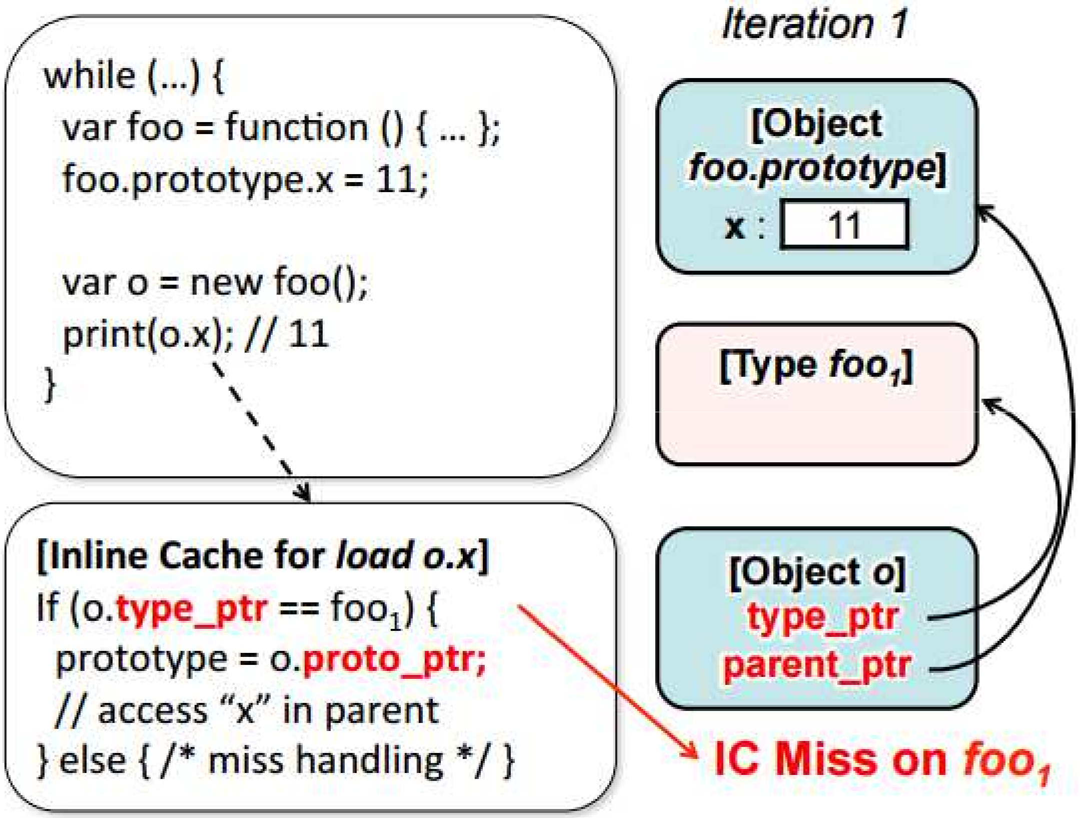
[Type *foo$_1$*]

[Object o]

type_ptr

parent_ptr

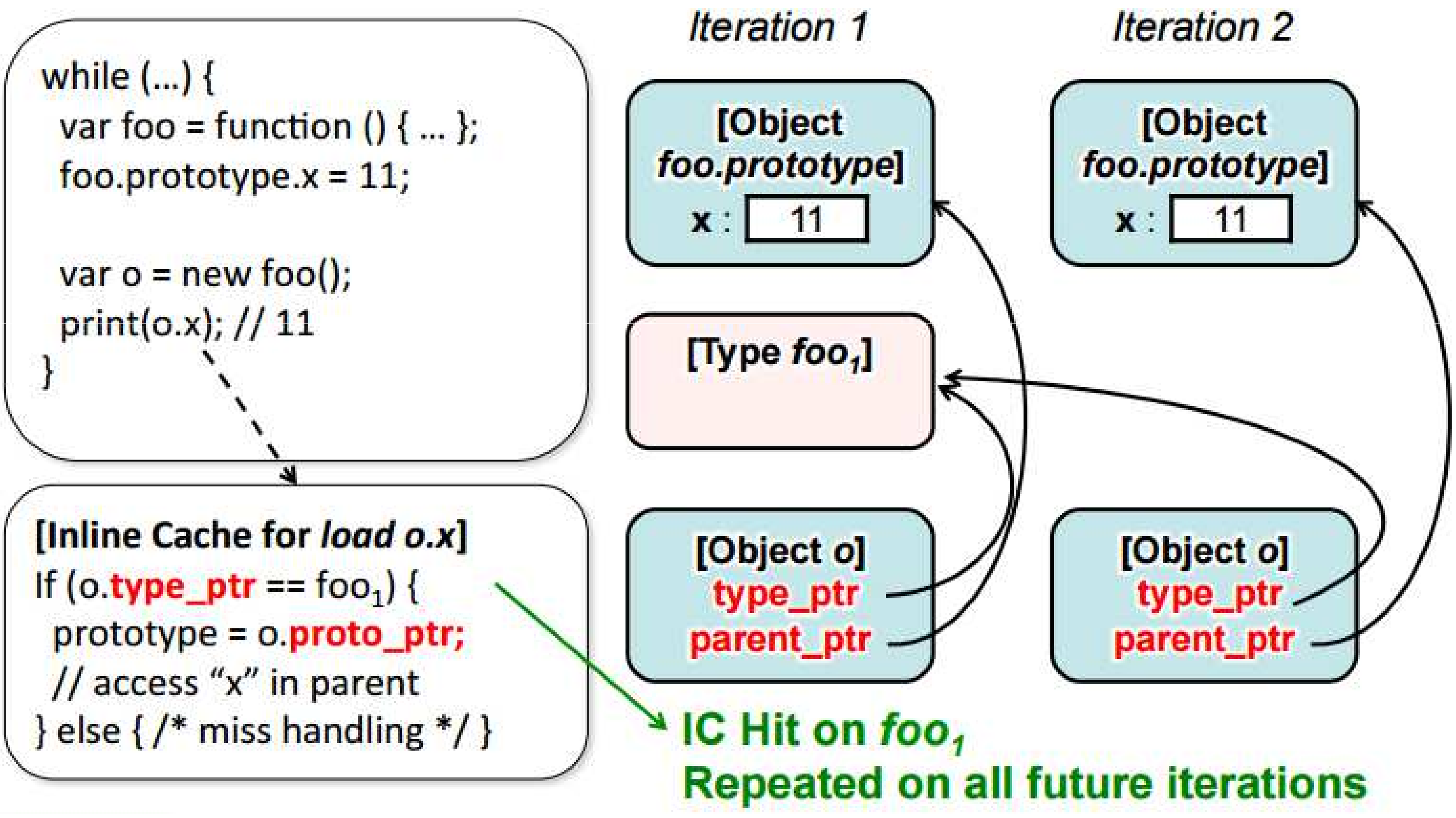
Solution: Decouple Inheritance from Types



```
while (...) {
  var foo = function () { ... };
  foo.prototype.x = 11;

  var o = new foo();
  print(o.x); // 11
}
```

```
[Inline Cache for load o.x]
If (o.type_ptr == foo1) {
  prototype = o.proto_ptr;
  // access "x" in parent
} else { /* miss handling */ }
```

Solution: Decouple Inheritance from Types

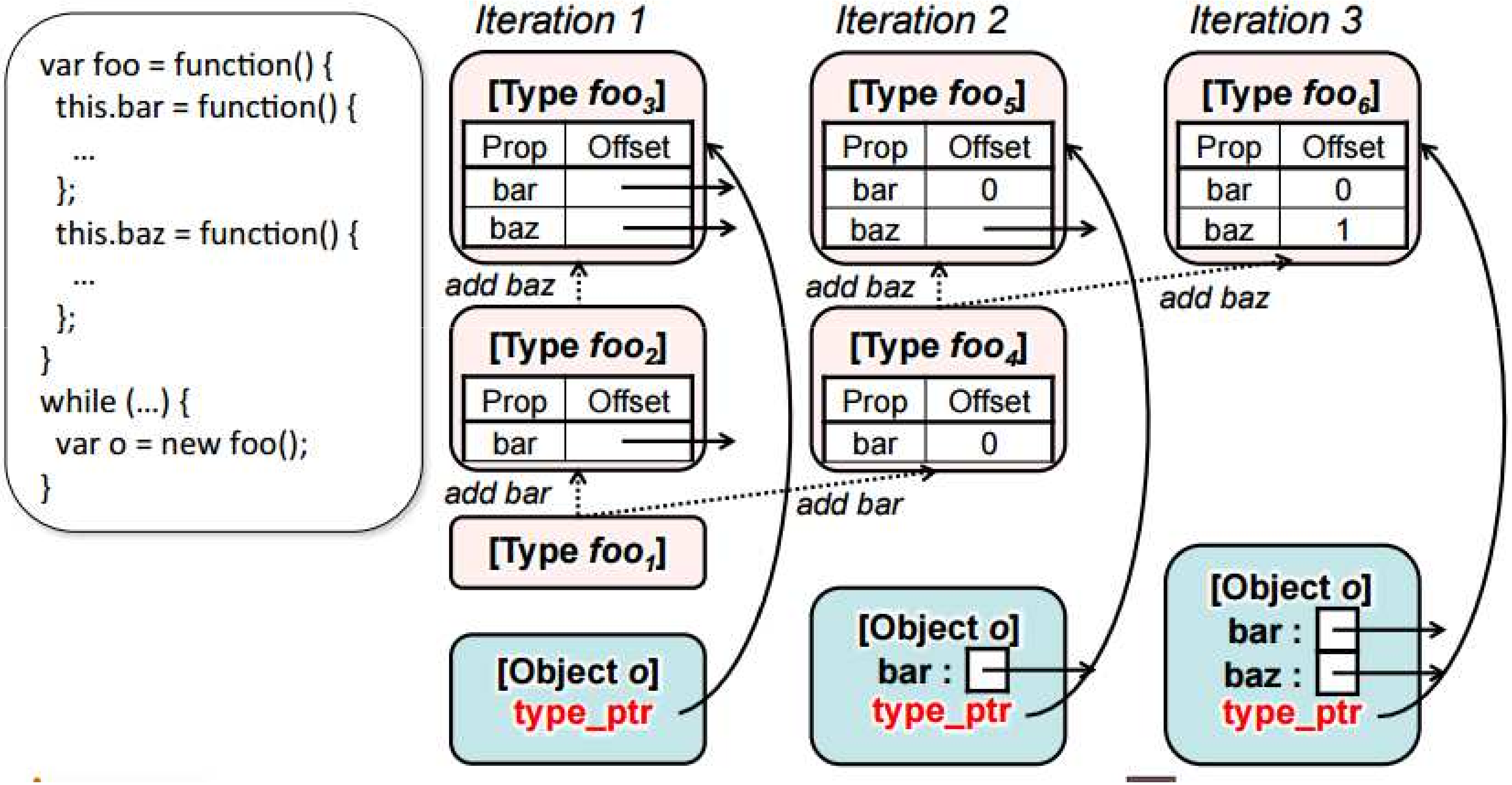


```
while (...) {
  var foo = function () { ... };
  foo.prototype.x = 11;

  var o = new foo();
  print(o.x); // 11
}
```

```
[Inline Cache for load o.x]
If (o.type_ptr == foo$_1$) {
  prototype = o.proto_ptr;
  // access "x" in parent
} else { /* miss handling */ }
```

IC Hit on *foo$_1$*
Repeated on all future iterations

Problem: Rampant Type Creation Due to Encoding Method Bindings



```
var foo = function() {
  this.bar = function() {
    ...
  };
  this.baz = function() {
    ...
  };
}
while (...) {
  var o = new foo();
}
```

[Type foo_3]

Prop	Offset
bar	
baz	

[Type foo_2]

Prop	Offset
bar	

[Type foo_1]

[Type foo_5]

Prop	Offset
bar	0
baz	

[Type foo_4]

Prop	Offset
bar	0

[Type foo_6]

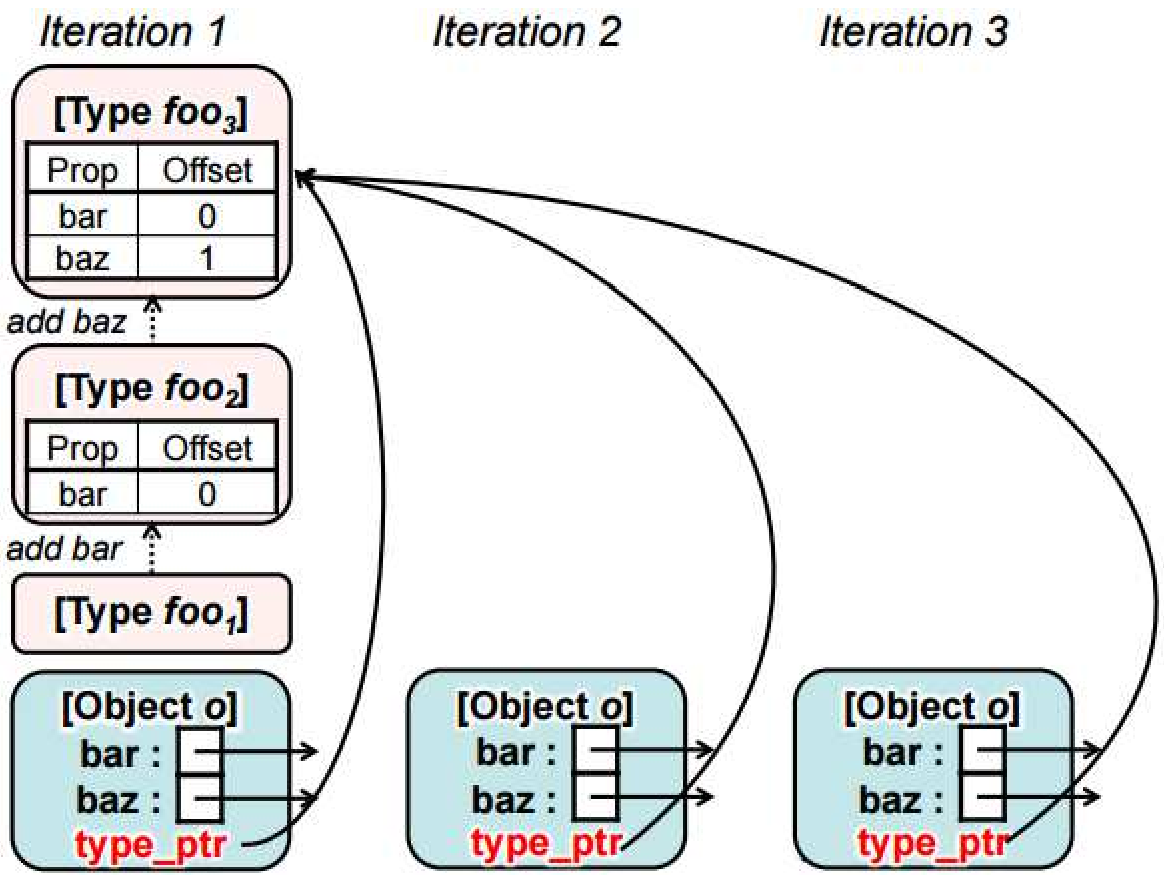
Prop	Offset
bar	0
baz	1

Problem: Rampant Type Creation Due to Encoding Method Bindings

- New type at end of each iteration, during the first N + 1 iterations - Where N is the number of function properties updated
- O(N2) type objects created in total

Solution: Decouple Method Bindings from Types

```
var foo = function() {
  this.bar = function() {
    ...
  };
  this.baz = function() {
    ...
  };
}
while (...) {
  var o = new foo();
}
```

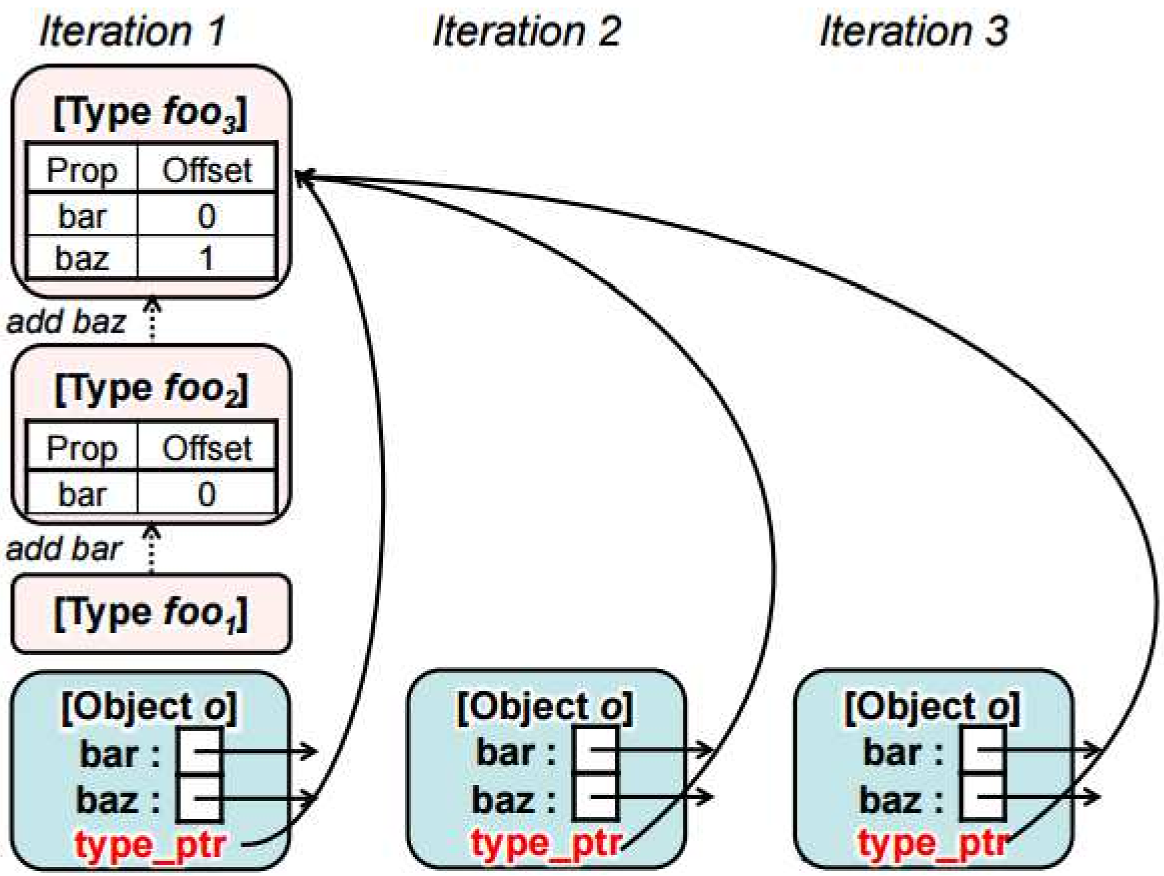


Prop	Offset
bar	0
baz	1

Prop	Offset
bar	0

Solution: Decouple Method Bindings from Types

```
var foo = function() {
  this.bar = function() {
    ...
  };
  this.baz = function() {
    ...
  };
}
while (...) {
  var o = new foo();
}
```



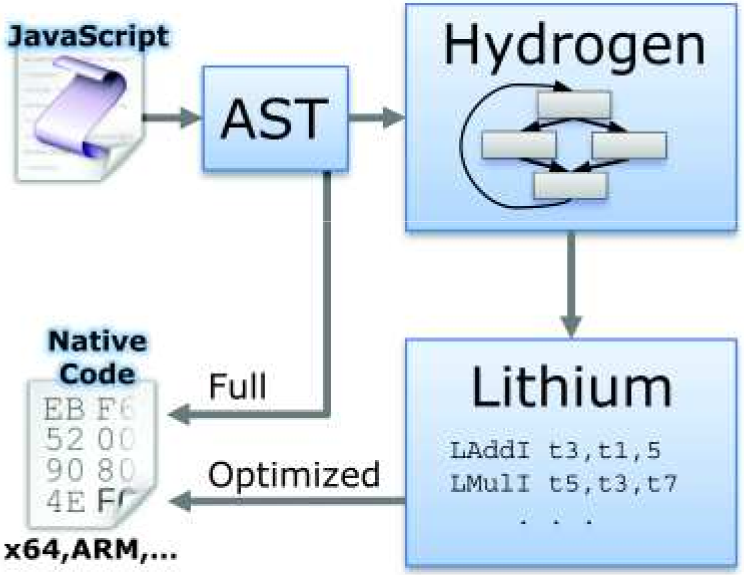
[Type foo_3]

Prop	Offset
bar	0
baz	1

[Type foo_2]

Prop	Offset
bar	0

Crankshaft

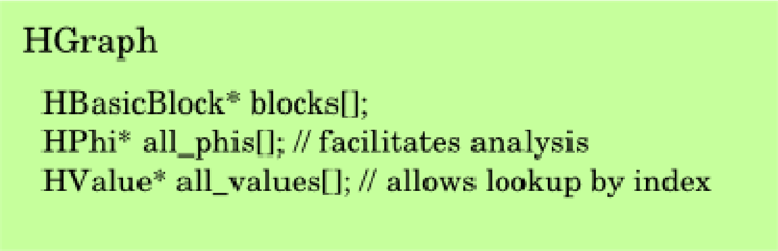

JavaScript
AST
Hydrogen
Native Code
EB F6
52 00
90 80
4E FC
x64,ARM,...
Full
Optimized
Lithium
LAddI t3,t1,5
LMulI t5,t3,t7
. . .

Hydrogen

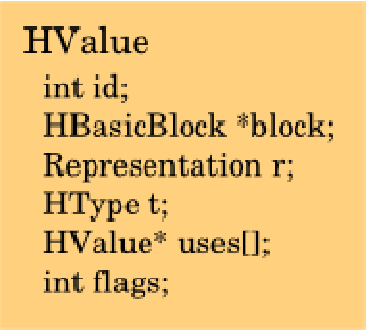
- Enable inlining
- Permit temporary values to be represented as untagged integer or double values
- To facilitate loop-invariant code motion and common subexpression elimination

SSA

Hydrogen consists of *values* - *instructions* and *phis* - contained in *basic blocks*, which themselves are contained in a *graph*.



```
HGraph
  HBasicBlock* blocks[];
  HPhi* all_phis[]; // facilitates analysis
  HValue* all_values[]; // allows lookup by index
```



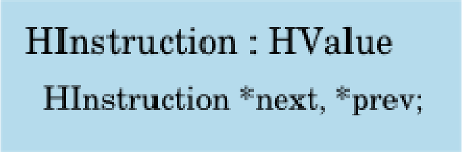
```
HValue
  int id;
  HBasicBlock *block;
  Representation r;
  HType t;
  HValue* uses[];
  int flags;
```

The standard SSA notation associates a named value with every expression:

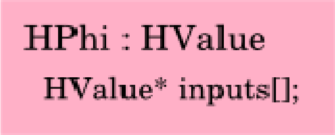
t1 = x + y;

t2 = 3;

t3 = t1 + t2; ...



```
HInstruction : HValue
  HInstruction *next, *prev;
```



```
HPhi : HValue
  HValue* inputs[];
```

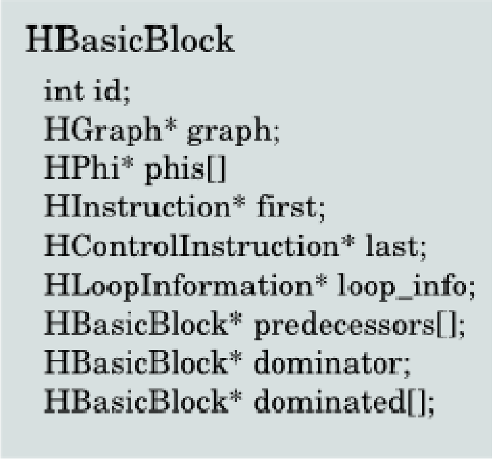
SSA

All values have a pointer to their containing block and to their *uses*.

Values also have a pointer to the next instruction in their block.

The last instruction in a block is a *control instruction*.

Every block has an array of pointers to its predecessors and successors, and also to its dominator block, and to all the blocks that it dominates.



```
HBasicBlock
  int id;
  HGraph* graph;
  HPhi* phis[]
  HInstruction* first;
  HControlInstruction* last;
  HLoopInformation* loop_info;
  HBasicBlock* predecessors[];
  HBasicBlock* dominator;
  HBasicBlock* dominated[];
```

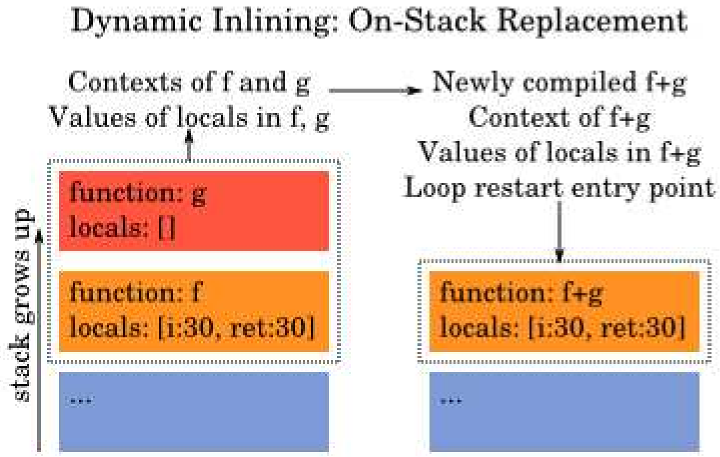
Crankshaft

- Not only does it have to optimize JavaScript, which is hard enough; it also has to do so in a way that permits on-stack replacement to optimize a long-running loop, without exiting the loop.

- Crankshaft needs to walk the abstract syntax tree (AST) to emit Hydrogen code *in the same way as the full compiler* - simulate what the full compiler is doing: what is on the stack, which variables are heap-allocated and which are not, and, for named local variables, what HValue is bound to that variable

OSR Example

```
function g () { return 1; }
function f () {
  var ret = 0;
  for (var i = 1; i < 10000000; i++)
  {
    ret += g ();
  }
  return ret;
}
```



Dynamic Inlining: On-Stack Replacement

OSR Example

```
0 push rbp                ;; Save the frame pointer.
1 movq rbp,rsp            ;; Set the new frame pointer.
4 push rsi                ;; Save the callee's "context object".
5 push rdi                ;; Save the callee's JSFunction object.
6 subq rsp,0x28           ;; Reserve space for 5 locals.

10 movq rax,rsi           ;; Store the context in a scratch register.
13 movq [rbp-0x38],rax ;; And save it in the last (4th) stack slot.
17 cmpq rsp,[r13+0x0]  ;; Check for stack overflow
21 jnc 28                 ;; If we overflowed,
23 call 0x7f7b20f40a00 ;; Call the overflow handler.

28 movq rax,[rbp+0x10] ;; Receiver object to rcx (unused).
32 movq rcx,rax           ;;
35 movq rdx,[rbp-0x38] ;; Global objectcontext to rdx.
39 movl rbx,(nil)         ;; Loop counter (i) to 0.
44 movl rax,(nil)         ;; Accumulator (ret) to 0.
49 jmp 97                 ;; Jump over some stuff.
```

OSR Example

```
Value f_and_g(Value receiver)
{
    // Stack slots (5 of them)
    Value osr_receiver, osr_unused, osr_i, osr_ret, arg_context;
    // Dedicated registers
    register int64_t i, ret;

    arg_context = context;

    // Assuming the stack grows down.
    if (&arg_context > root[STACK_LIMIT_IDX])
        // The overflow handler knows how to inspect the stack.
        // It can longjmp(), or do other things.
        handle_overflow();

    i = 0;
    ret = 0;
```

OSR Example

```
54  movq rax, rsi
57  movq rbx, [rbp - 0x28]    ;; load loop counter
61  testb rbx, 0x1
64  jnz 189  (0x7f7b20fa2a7d)
70  shrq rbx, 32
74  movq rdx, [rbp - 0x30]    ;; load acc
78  testb rdx, 0x1
81  jnz 237  (0x7f7b20fa2aad)
87  shrq rdx, 32
91  movq rcx, [rbp - 0x18]    ;; load receiver
95  xchgq rax, rdx
```

OSR Example

Check if the current definition of g, the function that we inlined below, is actually the same as when the inlining was performed.

```
97  movq rdx, [rsi + 0x2f]    ;; Slot 6 of the context : the global object.
101  movq rdi, 0x7f7b20e401e8 ;; Location of cell holding `g'
111  movq rdi, [rdi]          ;; Dereference cell
114  movq r10, 0x7f7b205d7ba1 ;; The expected address of `g'
124  cmpq rdi, r10            ;; If they're not the same...
127  jnz 371                  ;; Deoptimization bailout 2
```

OSR Example

```
osr_after_inlining_g:
    if (val_is_smi(osr_i))
        i = val_to_smi(osr_i);
    else
    {
        if (!obj_is_double(osr_i))
            goto deoptimize_3;

        double d = obj_to_double(val_to_obj(osr_i));

        i = (int64_t)trunc(d);
        if ((double)i != d || isnan(d))
            goto deoptimize_3;
    }
```

OSR Example

```
if (val_is_smi(osr_ret))
    ret = val_to_smi(osr_ret);
else
{
    if (!obj_is_double(osr_ret))
        goto deoptimize_4;

    double d = obj_to_double(val_to_obj(osr_ret));

    ret = (int64_t)trunc(d);
    if ((double)ret != d || isnan(d))
        goto deoptimize_4;

    if (ret == 0 && signbit(d))
        goto deoptimize_4;
}
goto restart_after_osr;
```

OSR Example

```
133  movq rdx, [rdx + 0x27] ;; Another redundant load.
137  cmpl rbx, 0x989680     ;; 10000000, you see.
143  jge 178                ;; If i >= 10000000, break.
149  movq rdx, rax          ;; tmp = ret
152  addl rdx, 0x1          ;; tmp += 1
155  jo 384                 ;; On overflow, deoptimize.
161  addl rbx, 0x1          ;; i++
164  movq rax, rdx          ;; ret = tmp
167  cmpq rsp, [r13 + 0x0]  ;; Reload stack limit.
171  jnc 137                ;; Loop if no interrupt,
173  jmp 306                ;; Otherwise bail out.
178  shlq rax, 32           ;; Tag rax as a small integer.
182  movq rsp, rbp          ;; Restore stack pointer.
185  pop rbp                ;; Restore frame pointer.
186  ret 0x8                ;; Return, popping receiver.
```

OSR Example

```
restart_after_osr:
    if (*g_cell != expected_g)
        goto deoptimize_1;

    while (i < 10000000)
    {
        register uint64_t tmp = ret + 1;
        if ((int64_t)tmp < 0)
            goto deoptimize_2;

        i++;
        ret = tmp;

        // Check for interrupt.
        if (&arg_context > root[STACK_LIMIT_IDX])
            handle_interrupt();
    }
    return val_from_smi(ret);
```

AST-to-Hydrogen

- AST-to-Hydrogen translation process handles phi insertion.

- So unlike the textbook dominance-frontier algorithm for optimal phi insertion, the HGraphBuilder simply adds phi values whenever a block has more than one predecessor, for every variable in the environment that is not always bound to the same value. For loops, a phi is added for every variable live in the environment at that time.

- A later pass attempts to prune the number of phis, where it can.

Type feedback

- AST has integer identifiers assigned to each node, which were are embedded in the inline caches used by the full-codegen code. This correspondence allows crankshaft to know the types of the values that have been seen at any given call site, property access, etc.

- This information is used by the graph builder to initialize the typefield of each HValue, to help in type inference.

inlining

- When the graph builder reaches a call, it can try to inline it. Note that this happens very early, before any optimizations are made on the source code.

Some conditions:

1. The function to be inlined is less than 600 characters long.
2. Neither the inner nor the outer functions may contain heap-allocated variables. (In practice, this means that neither function may contain lexical closures.)
3. Inlining of for-in, with, and some other expression types is not currently supported.
4. There is a limit on the maximum inlining depth.
5. A function's call to itself will not be inlined.
6. There is a limit to the number of nodes added to the AST.

Source-to-source optimization passes

- As parsing proceeds, Crankshaft generates specific HInstruction nodes and packs them into HBasicBlocks.

- There is a large set of about 120 instructions

- At this point, inlining is complete, so Crankshaft moves to focus on representation of temporary values. The goal is to unbox as many values as possible, allowing the use of native machine instructions, and decreasing the allocation rate of tagged doubles. The Hydrogen SSA language directly facilitates this control-flow and data-flow analysis.

Source-to-source optimization passes

- Block preordering
  Crankshaft reorders the array of blocks in the HGraph to appear in reverse post-order, equivalent to a topological sort. In this way, iterating the array of blocks from beginning to end traverses them in data-flow order.

- dominator assignment
  The reverse post-order sort from the previous step facilitates calculation of the dominator for each block. The entry block dominates its successors. Other blocks are iterated over, in order, assigning them the dominator of their predecessors, or the common dominator of their predecessors if there is more than one.

Source-to-source optimization passes

- **mark dead subgraphs**
  - It can often be the case that a function being optimized does not have full type information. If the parser reaches such a branch, then it inserts an unconditional soft deoptimization, to force the collection of more type information.
  - To avoid the compiler wasting its time trying to optimize loops and move code in parts of the graph that have never been reached, this dead-subgraph pass puts a mark on all blocks dominated by a block containing an unconditional soft deoptimization. As you can see, this analysis is facilitated by the SSA dominator relation.

Source-to-source optimization passes

- **redundant phi elimination**
  - This pass eliminates phi values whose input is always the same, replacing the uses with the use of the value.
  - All phis are placed on a work list. Then, while there are any phis on the worklist, a phi is taken off the list, and if it is eliminatable, it is replaced. Now here's the trick: if the phi was in turn used by any other phi, the phi uses are pushed onto the list, if they weren't there already. Iteration proceeds until the worklist is empty.

- **dead phi elimination, and phi collection**
  - Some phi values are dead, having no real uses, neither direct nor indirect (through some other phi). This pass eliminates any phi that doesn't have a real use. It's somewhat like a garbage collection algorithm, and also uses a worklist.

As the phis are collected, if any undefined value is found to reach a phi, or arguments can reach a phi only via some inputs, then optimization is aborted entirely.

Source-to-source optimization passes

- **representation inference**
  - HValue has a type field and a representation field
  - The goal is to represent temporaries as untagged int32 or double values.
  - For each phi, if any operand of the phi or any connected phi may not be converted to an int32, Crankshaft pessimistically marks the phi as not convertible to an int32.
  - Finally, for all values with flexible representation - phis, Math.abs, and binary bitwise and arithmetic operations - Crankshaft determines the representation to use for that value.
  - If the representation can be inferred directly from the inputs, then that representation is used. Otherwise, some heuristics are run on the representations needed by the use sites of the value.

Source-to-source optimization passes

- **Range analysis**
  - This pass looks at each value, and tries to determine its range, if Crankshaft was actually able to allocate a specialized representation for it.
  - In practice this is mainly used for values represented as integers. It allows HInstructions to assert various properties about the instruction, such as lack of overflow, or lack of negative zero values.

- **Type inference and canonicalization**
  - Representation is about how to use an object; type is about the object itself.
  - Type inference can help eliminate runtime checks.
  - After type inference is run, each instruction in the whole graph is asked to canonicalize itself, which basically eliminates useless instructions: HToInt32 of a value with int32 representation, HCheckNonSmi of a value with non-smi type, etc.

Source-to-source optimization passes

- **representation changes**
  - Up to this point we've been dealing with values on a fairly high level, assuming that all of the definitions will be compatible with the uses. But as we chose to represent some temporaries as untagged values, there will need to be explicit conversions between representations -- hopefully not too many, but they will be needed. This pass inserts these representation changes.
  - This pass uses the truncating-to-int32 and deoptimize-on-undefined flags calculated in previous phases, to tell the changes what conditions they need to watch out for. Representation changes for phis happen just before branching to the block with the phi -- effectively, just before calling the block.
- **minus zero checks**
  - The previous pass probably inserted a number of HChange instructions. For any HChange from int32 (which must be to tagged or double), Crankshaft traces dataflow backwards to mark the HChange(s) that produced the int32 as deoptimizing on negative zero.

Source-to-source optimization passes

- **stack check elimination**
  - Loops need to be interruptable, and V8 does so by placing a stack check at the beginning of each loop iteration. If the runtime wants to interrupt a loop, it resets the stack limit of the process, and waits for the process's next stack check.
  - However if a call dominates all backward branches of a loop, then the loop can be interrupted by the stack check in the callee's prologue, so the stack check in the loop itself is unneeded and can be removed.

Source-to-source optimization passes

At this point, Hydrogen has served two of its three purposes: inlining and unboxing. The third is loop-invariant code motion (LICM) and common subexpression elimination (CSE). These important optimizations are performed by the perhaps misnamed global value numbering (GVN) pass.

- **global value numbering: licm**
  - LICM. For every instruction in a loop, if it only depends on values defined outside the loop, and the side effects of the loop as a whole do not prevent the move, then LICM hoists the instruction to the loop header.
  - Computes the side effects for the entire loop, and moves any instructions that it can to the loop header.

Source-to-source optimization passes

- **global value numbering: cse**
  - A common subexpression is a Hydrogen value B that is "equal" in some sense to a previous instruction A, and whose value will not be affected by any side effects that occur on the control-flow paths between A and B.
  - "value numbering" essentially means "define appropriate hash and equality functions and stick all the instructions you've seen in a hash table.“
  - As GVN goes through the instructions of each block, in reverse-post-order, it puts them into this hash table (HValueMap). When GVN reaches an instruction that causes side effects, it flushes out the entries that depend on those side effects from the table. In that way, if ever it sees an instruction B that is equal to an instuction A in the table, GVN can remove B from its block, and replace every use of B with a use of A.
  - Global value numbering works across basic blocks, hence the "global" in its name.

Source-to-source optimization passes

- **checked value replacement**
  - In optimized code, accessing an array out of its bounds will cause a deoptimization.
  - AST-to-Hydrogen translation inserts a bounds check, and has the array access use the result of the bounds check. But the purpose of this instruction is just to cause the side effect of deoptimization, and the resulting value is just a copy of the incoming index, which increases register allocation pressure for no reason.
  - Crankshaft replaces uses of HBoundsCheck with the instruction that defines the index value, relying on the HBoundsCheck's position in the instruction stream to cause deopt as needed.