PPLサマースクール2021

JavaScript エンジンの実装技術

東京大学大学院情報理工学系研究科

鵜川 始陽

自己紹介

- 鵜川始陽（東京大学大学院情報理工学系研究科）
  - 2005 京都大学
  - 2008 電気通信大学
  - 2014 高知工科大学
  - 2020 東京大学
- 専門: ごみ集め
  - 「ガベージコレクション
    自動メモリ管理を構成する理論と実装」翔泳社
- JavaScriptの研究
  - PPL 2010の招待公演を聞いて研究を始める
    - 小野寺民也 (IBM) 「オブジェクト指向言語余話」
  - IoT向けJavaScriptインタプリタeJSを開発
    https://github.com/tugawa/ejs-new.git

The garbage collection handbook (R.E.Jones他) の翻訳

JavaScript

- Webブラウザ内で動くプログラミング言語
  - 時計，カレンダー
  - AJAX: Google Mapなど
    - ページ遷移せずにサーバからコンテンツを取得
  - Webフレームワーク: Ruby on Railsなど
  - クラウドアプリ: G suite, Office365など
    - リモートワークで需要急増
- ブラウザ以外でも利用
  - サーバサイド: node.js
  - IoT: Espruino, JerryScript, eJS
- 世界一使われている言語
  - ピークアウトしている？

JavaScriptの歴史

Allen Wirfs-Brock & Brendan Eich: **JavaScript: The First 20 Years**, History of Programming Language (HOPL IV), 2021

- Netscape communications社が開発 1995/5
- ECMAで標準化: ECMA-262

略称	edition	公開日
ES1	1st	1997/6
ES2	2nd	1998/8
ES3	3rd	1999/12
ES4	4th	中断
ES5 (ES3.1)	5th	2009/12
ES5.1	5.1	2011/6
ES6 (ES20015)	6th	2015/6
ES2016	7th	2016/6

この辺色々
あったらしい

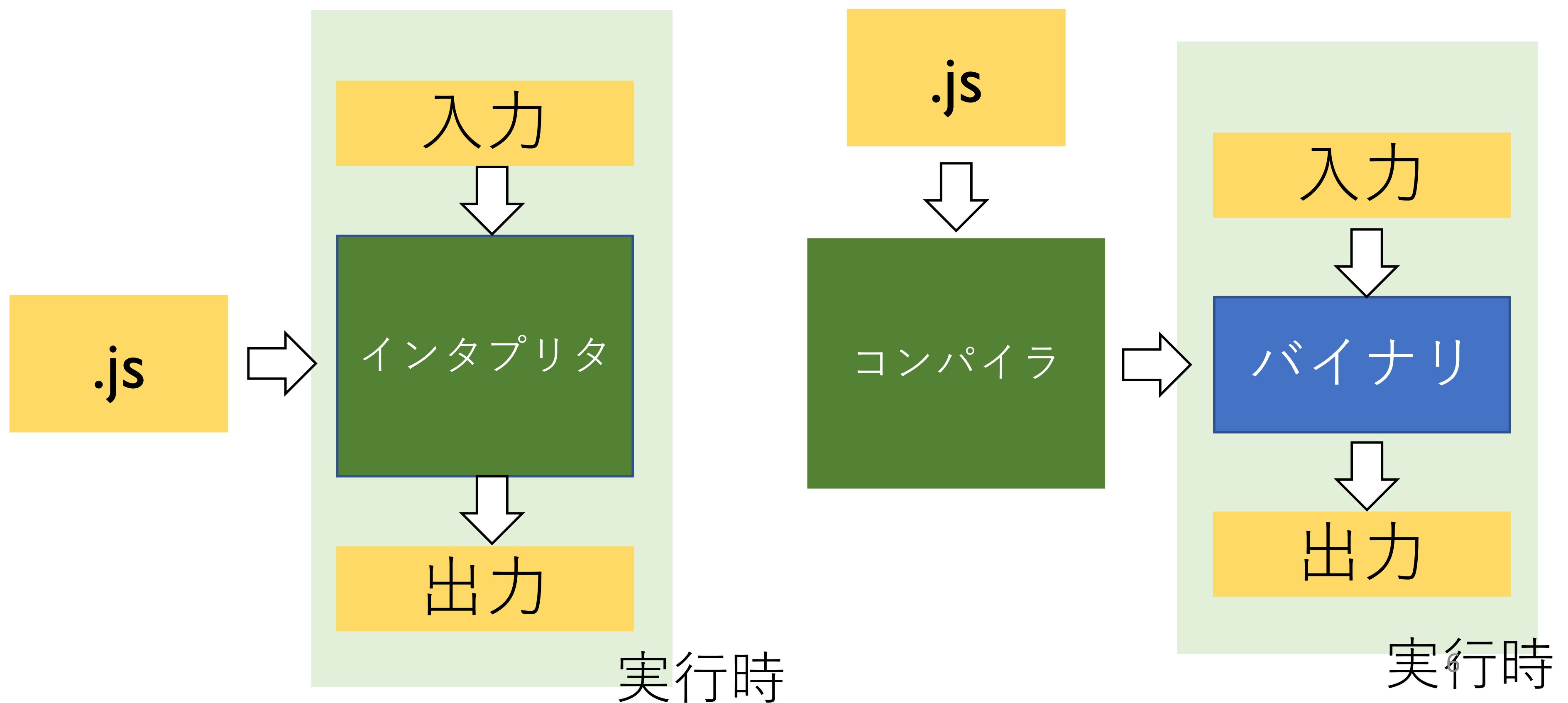
以降毎年更新

glue言語から汎用言語に

- 初期 (2000年ごろ)
  - C言語で書かれたのコンポーネントをつなぎ合わせるのりの役割
  - 性能はあまり気にならない
- 現在
  - クラウドアプリを記述する言語
  - サーバサイド、デスクトップアプリ
  - 組込みシステム
  - 性能はすごく気になる
    実行速度、実行に必要なメモリ

コンパイルによる高速化

- コンパイルすれば高速に実行できる？
  - LLVMに最適化させる



静的コンパイルの例

```
while(i < a.length) {sum += a[i]; i += 1;}
```

質問：どんなバイナリに
　　　コンパイルすれば良い？

例の出典：Manuel Serrano: **JavaScript AOT compilation**, DLS '18, 2018

静的コンパイルの例

```
while(i < a.length) {sum += a[i]; i += 1;}
```

```
B:cmp %i, 0(%a) ; length
  jge E
  add %sum, 4(%a, %i, 4)  ; %a+%i*4+4
  inc %i
  jmp B
E:
```

	length	[0]	[1]	[2]	[3]	[4]
a	5	3	1	7	10	5

例の出典：Manuel Serrano: **JavaScript AOT compilation**, DLS '18, 2018

静的コンパイルの例

```
while(i < a.length) {sum += a[i]; i += 1;}
```

```
B:cmp %i, 0(%a) ; length
  jge E
  add %sum, 4(%a, %i, 4)  ; %a+%i*4+4
  inc %i
  jmp B
E:
```

	length	[0]	[1]	[2]	[3]	[4]
a	5	3	1	7	10	5

静的コンパイルの例

```
var i = “0”
var a = {length: “011”,
         “0”: 0, “01”: 1, “011”: 2};
while(i < a.length) {sum += a[i]; i += 1;}
```

例の出典：Manuel Serrano: **JavaScript AOT compilation**, DLS ‘18, 2018

JavaScript言語の特徴

めちゃくちゃフレキシブル
=> 実行時まで分からないことが多い

- 動的型
- オブジェクトは連想配列
- オーバロードされた演算子+自動型変換
  - ユーザ定義は不可 (vs. ruby, C++)

動的型

- 値が型を持つ
- 式や変数には型がない
  - 様々な型の値が格納される可能性がある

```
function f(a, b) {
  if (a === 0)
    return b;
  return a;
}
f(0, “abc”); // => “abc”
f(“abc”, 1); // => “abc”
f(1, “abc”); // => 1
```

JavaScript(ES5)の型
- 数値
- 文字列
- 真偽値
- undefined
- オブジェクト

オブジェクトは連想配列

- オブジェクト＝プロパティ名→値のマップ
  - プロパティには型がない（値に型がある）
  - 実行中にプロパティが増える
  - プロパティ名は任意の文字列
  - 実行時にプロパティ名を計算可能
    - []を使ったアクセス

```
var obj = {“x”: “abc”}
obj.x = 0;            //{“x”:0}
obj.y = 1;            //{“x”:0,“y”:1}
var n = “z”;
obj.[n + “1”] = 5; //{“x”:0,“y”:1,“z1”: 5}
```

メソッド

- メソッド = プロパティに格納された関数
  - thisは呼び出される時に決まる

```
function f() {
  return this.x;
}
obj = {"x": 1};
obj.getX = f;  // obj = {"x":1, "getX":関数}
obj.getX();    // => 1
```

演算子オーバーロード

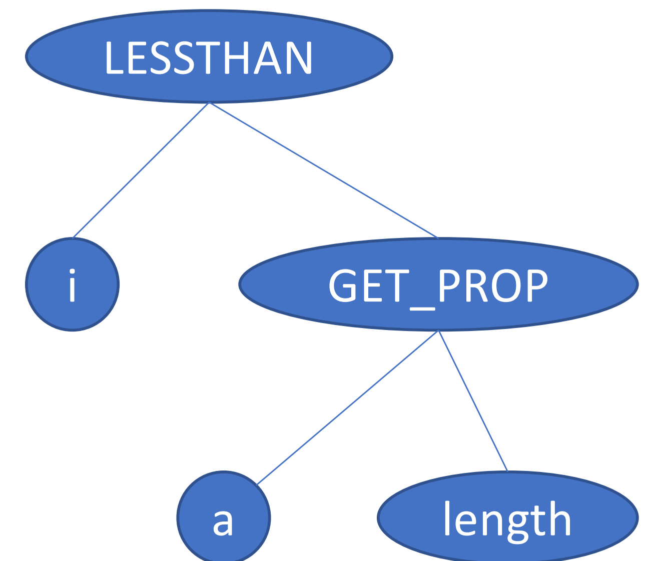
- 同じ演算子でもオペランドの型で動作が変わる
- オペランドの型は自動で変換される

```
1 + 1;      // => 2
”1” + “1”; // => “11”
1 + “1”;   // => “11”
function add(a, b) {
 return a + b; // オペランドが即値でない
}
add(1,2); // => 3
add(true, “abc”); // => “trueabc”
```

インタプリタ

```
while(i < a.length) {sum += a[i]; i += 1;}
```

```
eval(n) {  //nは構文木の節点
  …
  if(n->type == LESSTHAN){
    l = eval(n->lhs);
    r = eval(n->rhs);
    return op_lt(l, r);
  }
}
```



AST(構文木)

インタプリタ

```
while(i < a.length) {sum += a[i]; i += 1;}
```

```
ev
}
```

```
op_lt(lhs, rhs) {
  if (type(lhs)==OBJ) {lhs=lhs.valueOf()}
  if (type(rhs)==OBJ) {rhs=rhs.valueOf()}
  if (type(lhs)==STR && type(rhs)==STR)
    return strcmp(lhs, rhs) < 0;
  if (type(lhs)!=NUM) lhs=to_num(lhs);
  if (type(rhs)!=NUM) rhs=to_num(rhs);
  if (type(lhs)!=NUM || type(rhs)!=NUM)
    return FALSE;
  return lhs < rhs;
}
```

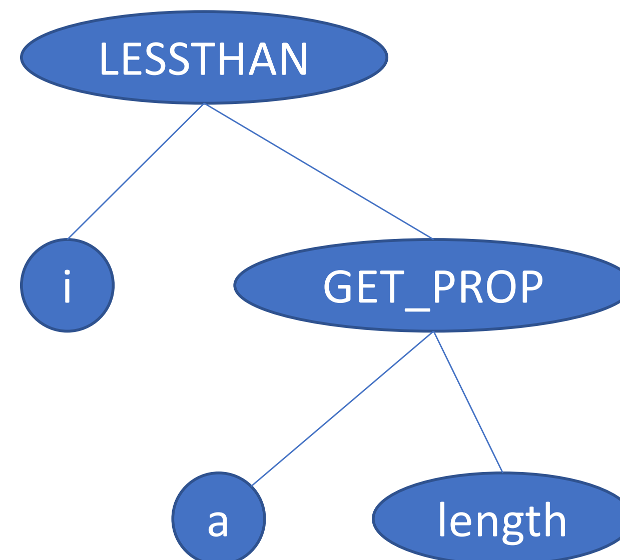
- 型によるディスパッチ
- 型変換

※正確にはもっと複雑な処理が必要です

インタプリタ

```
while(i < a.length) {sum += a[i]; i += 1;}
```

```
eval(n) {
   …
   if(n->type == GETPROP){
     o = eval(n->obj);
     p = eval(n->prop);
     if (type(p) != STR)
       p = to_str(p);
     // 連想配列の検索
     return lookup(o, p);
   }
}
```



静的コンパイルの例

```
while(i < a.length) {sum += a[i]; i += 1;}
```

```
B: i = get_global(“i”);
   len = lookup(a, “length”);
   tmp = op_lt(i, len);
   if (tmp == FALSE) goto E;
   if (type(i) != STR)
     tmp = lookup(a, to_str(i));
   else
     tmp = lookup(a, i);
   sum = op_add(sum, tmp);
   set_global(“i”, op_add(i, 1));
   goto B;
E:
```

静的コンパイルでは

簡単には性能が出ない

- 重いところはコンパイルしても残る
  - フィールドアクセス
    - 実行時に計算されるフィールド名
    - 複雑な検索　(vs. C言語のオフセットアクセス)
  - 演算子
    - オペランドの値の型でディスパッチ
    - 型の変換
- インライン展開できない => 最適化が効かない
  - 呼び出すメソッドの実体は実行時に決まる
    （メソッドはプロパティの値）

静的コンパイルの研究

Manuel Serrano: **JavaScript AOT compilation**, DLS '18, 2018

楽観的な仮定：
The most likely data structure a program will use is the one for which the compiler is able to produce its best code.
（現実のプログラムではコンパイラに優しいデータ構造を扱うことが多い）

二つのバージョンにコンパイル

if (楽観的な仮定が成立)

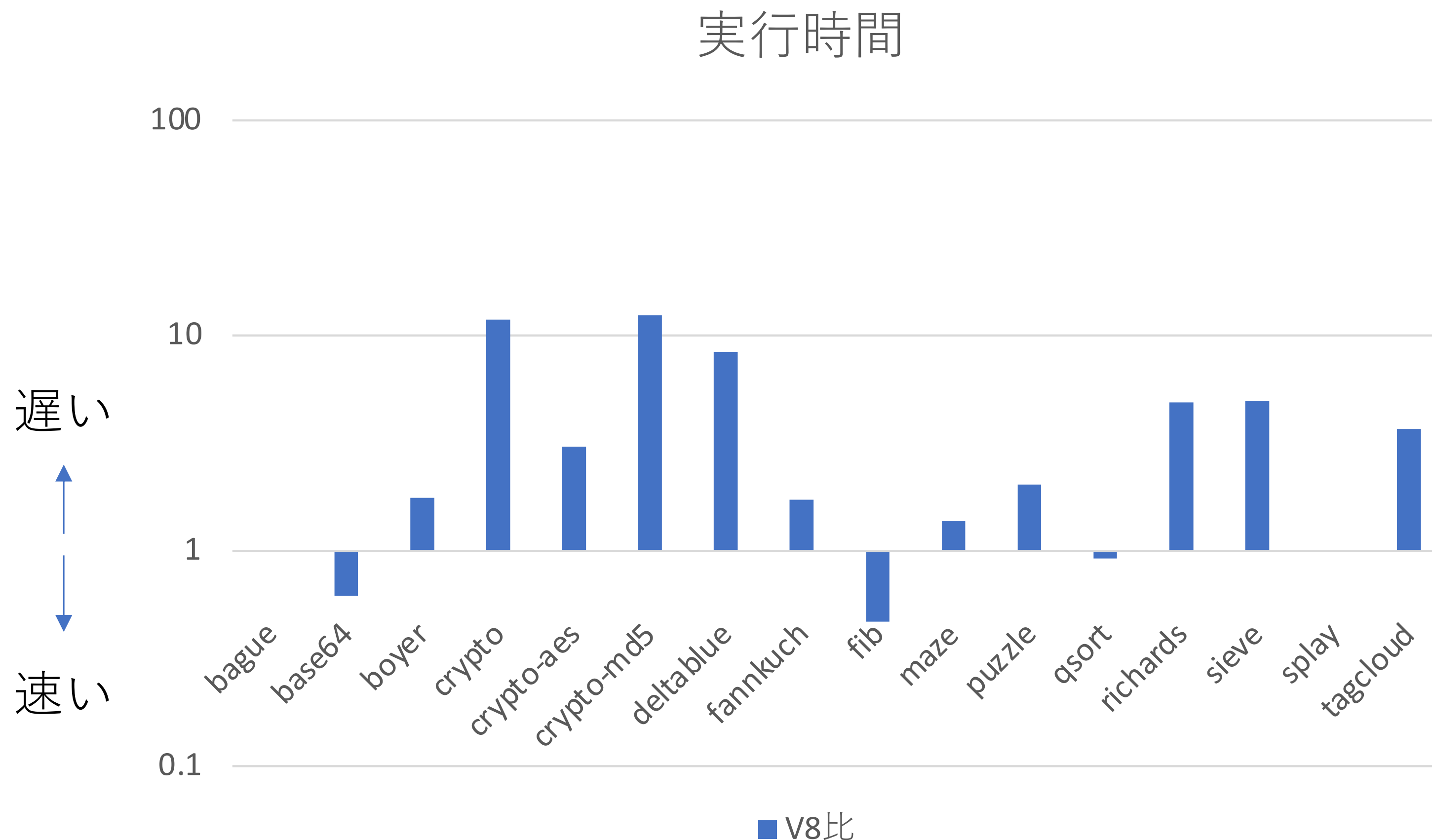
```
B:cmp %i, 0(%a) ; length
  jge E
  add %sum, 4(%a, %i, 4)  ; %a+%i*4+4
  inc %i
  jmp B
E:
```

else

```
B: i = get_global(“i”);
   len = lookup(a, “length”);
   tmp = op_lt(i, len);
   if (tmp == FALSE) goto E;
   if (type(i) != STR)
     tmp = lookup(a, to_str(i));
   else
     tmp = lookup(a, i);
   sum = op_add(sum, tmp);
   set_global(“i”, op_add(i, 1));
   goto B;
E:
```

結果

- 最新のインタプリタ+JITには及ばない
  - 目標：インタプリタ(+JIT)の半分の性能
  - ベンチマークではV8の0.47倍～12.45倍の実行時間



JITコンパイル（実行時コンパイル）

- 実行時にプロファイリングした情報を利用
  - 変数の型
  - オブジェクトの型
    - プロパティ名、プロパティの型
- データ構造もそれに合わせて最適化
  - オブジェクト: 連想配列 -> 構造体
- 主要なアイデアはコンパイルするかどうかとは独立
  - hidden class, inline caching

静的言語の構造体（C言語）

- 定数オフセットアクセスにコンパイル
- フィールドのオフセットは型とフィールド名から一意に決まる

obj

+0(x)	1
+4(y)	5

```
struct S {
  int x, y;
};
…
S* obj;
```

```
;; obj->x = 1
mov   0(%obj), 1
;; obj->y = 5
mov   4(%obj), 5
```

hidden class

オブジェクトに型をつける

JavaScriptのオブジェクト（再掲）

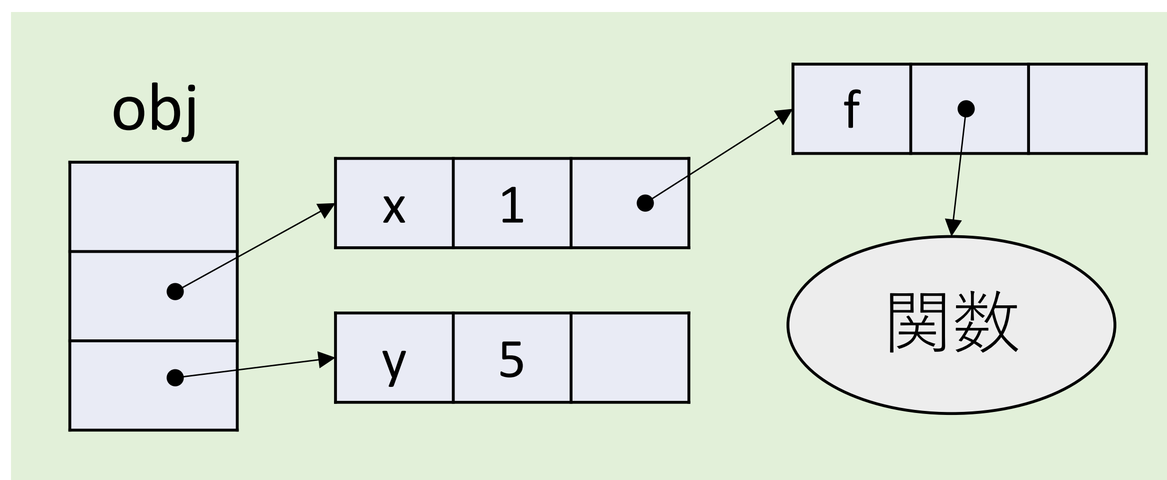
- オブジェクト＝プロパティ名→値のマップ
  - プロパティには型がない（値に型がある）
  - 実行中にプロパティが増える
  - プロパティ名は任意の文字列
  - 実行時にプロパティ名を計算可能
    - []を使ったアクセス

```
var obj = {“x”: “abc”}
obj.x = 0;            //{“x”:0}
obj.y = 1;            //{“x”:0,“y”:1}
var n = “z”;
obj.[n + “1”] = 5; //{“x”:0,“y”:1,“z1”: 5}
```

ハッシュ表による実装

- 全部ハッシュ表のlookup
  - メソッド呼び出し
  - プロパティアクセス
  - 配列の要素アクセス
- 遅い

```
var obj = {"x": 1};
obj.y = 5;
obj.f =function(){…};
```

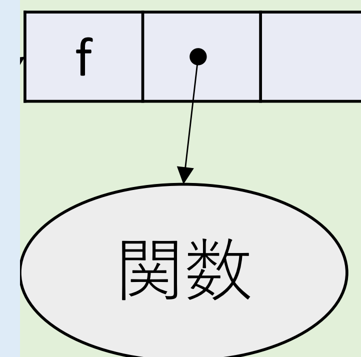


ハッシュ表による実装

- 全部ハッシュ表のlookup
  - メソッド呼び出し
  - プロパティアクセス
  - 配列の要素アクセス
- 遅い

```
var obj = {"x": 1};
obj.y = 5;
obj.f =function(){…};
```

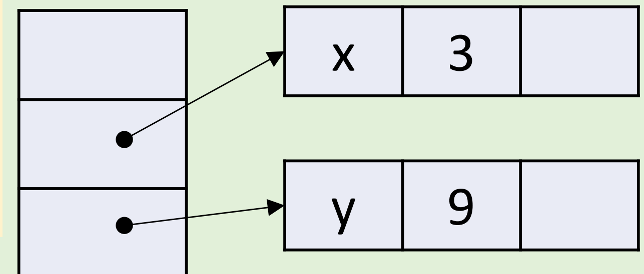
```
eval(n) {
  …        // obj.prop
  if(n->type == GET_PROP){
    o = eval(n->obj);
    p = eval(n->prop);
    loc = hash_lookup(o, p);
    return loc.value;
  }
}
```

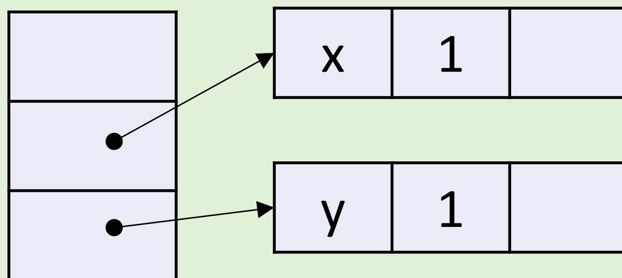


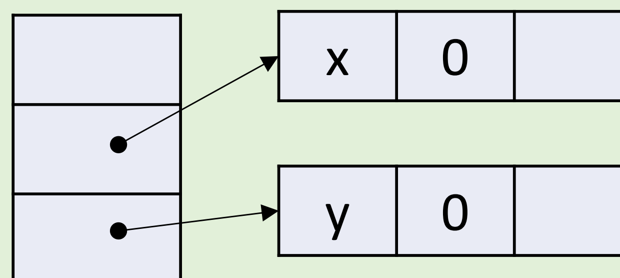
メモリの無駄も多い

- プロパティ名が重複する
- 検索のための仕組みにメモリが使われる

```
for (i = 0; i < N; i++) {
  var obj = {“x”: i, “y”: i*i};
  array.push(obj);
}
```







JavaScriptの全力

- 動的言語の柔軟性を全力で活用するプログラムにはこれが正解
  - 例）オブジェクトを辞書として使う

```
dict = {};
function put(key, value) {
  dict[key] = value;
}
function get(key) {
  return dict[key];
}
```

実際のプログラムは希望が持てる

- 似たオブジェクトは多く作られやすい
  - 同じところで作られるオブジェクトは同じプロパティを持つことが多い
  - 同じコンストラクタで初期化されると同じプロパティを持つ

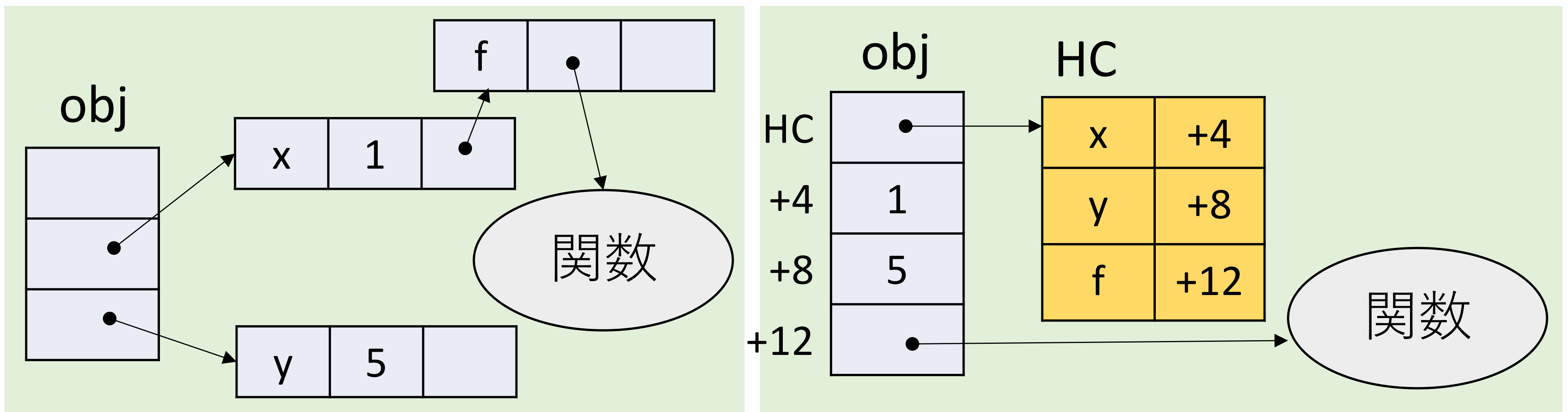
```
for (i = 0; i < N; i++) {
  var obj = {“x”: i, “y”: i*i};
  array.push(obj);
}
```

```
function Point(x, y) {
  this.x = x;
  this.y = y;
};
var p = new Point(1,2);
```

Chambers, C., Ungar, D. & Lee, E.: **An Efficient Implementation of SELF a Dynamically-Typed Object-Oriented Language Based on Prototypes**, OOPSLA ‘89, 1989

hidden class

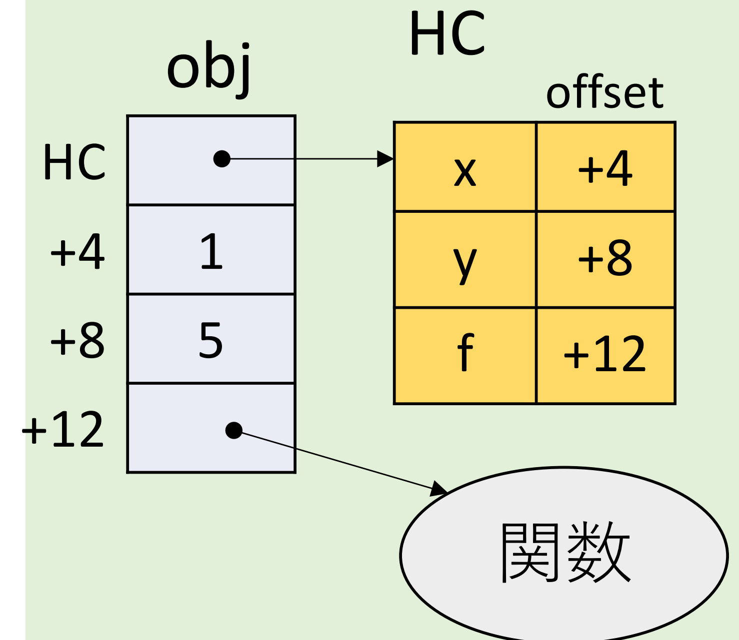
- オブジェクトのデータとレイアウトを分離
  - hidden class (HC): レイアウトを表すメタオブジェクト
  - HCがオブジェクトの型
- プロパティのアクセス
  1. HCを検索してオフセットを得る
  2. オブジェクト先頭アドレス+オフセットでアクセス



プロパティのアクセス

1. HCを検索してオフセットを得る
   - ここはハッシュ表のlookup
2. オブジェクト先頭アドレス+オフセットでアクセス

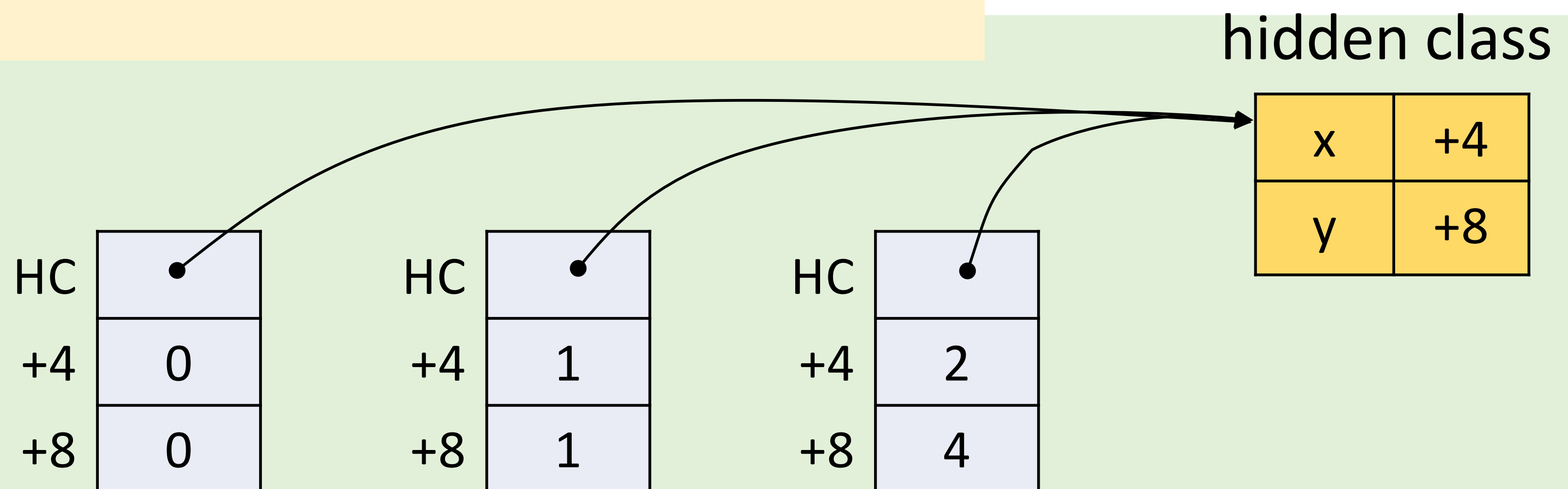
```
eval(n) {
  …        // obj.prop
  if(n->type == GET_PROP){
    o = eval(n->obj);
    p = eval(n->prop);
    loc = lookup(o->HC, p);
    return *(o + loc.offset);
  }
}
```



hidden classの共有

- 同じプロパティ集合を持つオブジェクトはHCを共有できる

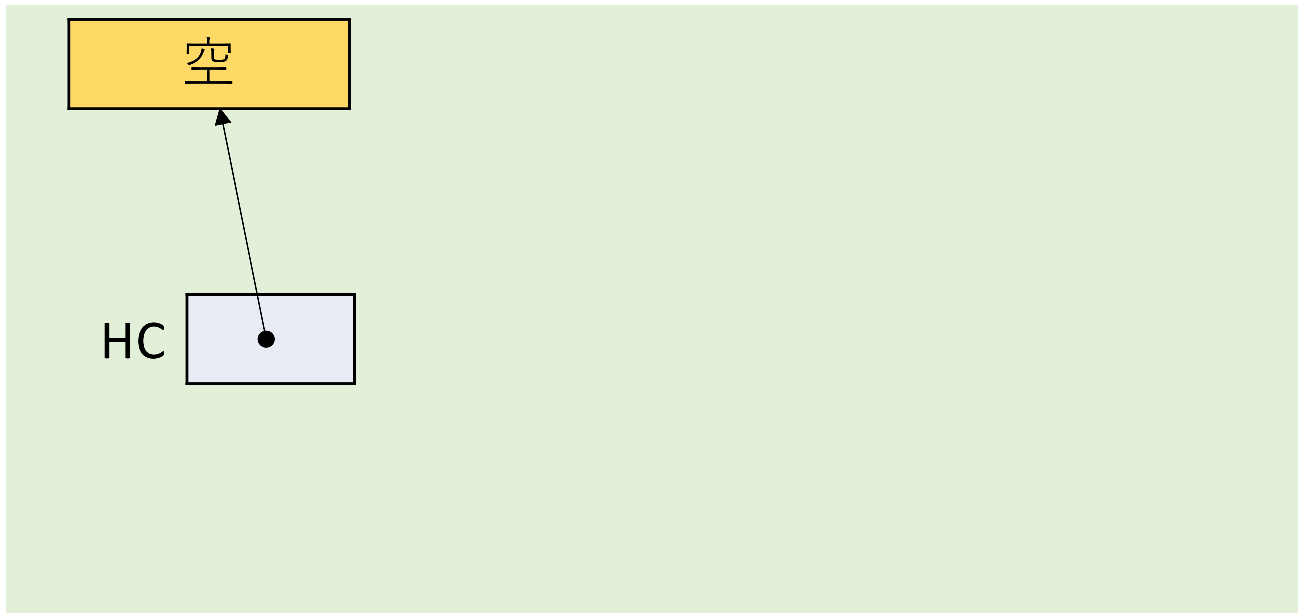
```
for (i = 0; i < N; i++) {
  var obj = {“x”: i, “y”: i*i};
  array.push(obj);
}
```



プロパティの追加

```
var obj = {};
obj.x = 1;
obj.y = 2;
```

- プロパティが増えるとHCが変わる
- 共有している可能性があるのでHCはread only



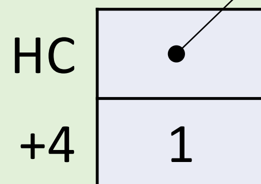
プロパティの追加

```
var obj = {};
obj.x = 1;
obj.y = 2;
```

- プロパティが増えるとHCが変わる
- 共有している可能性があるのでHCはread only

hidden class 0

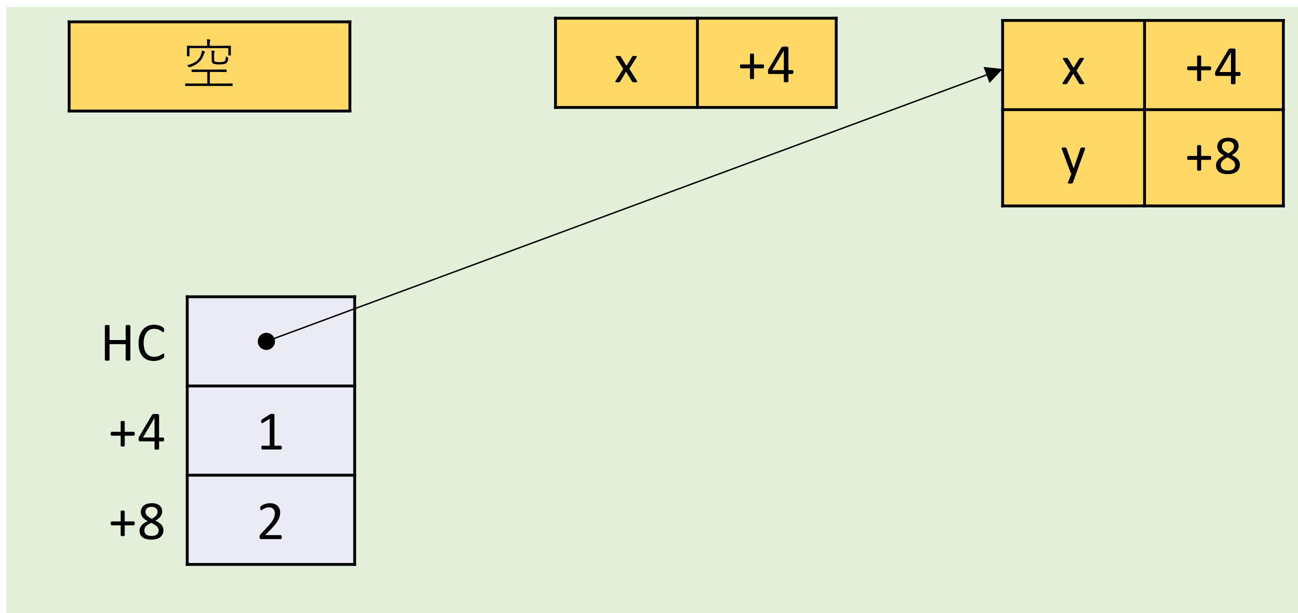
hidden class 1



プロパティの追加

```
var obj = {};
obj.x = 1;
obj.y = 2;
```

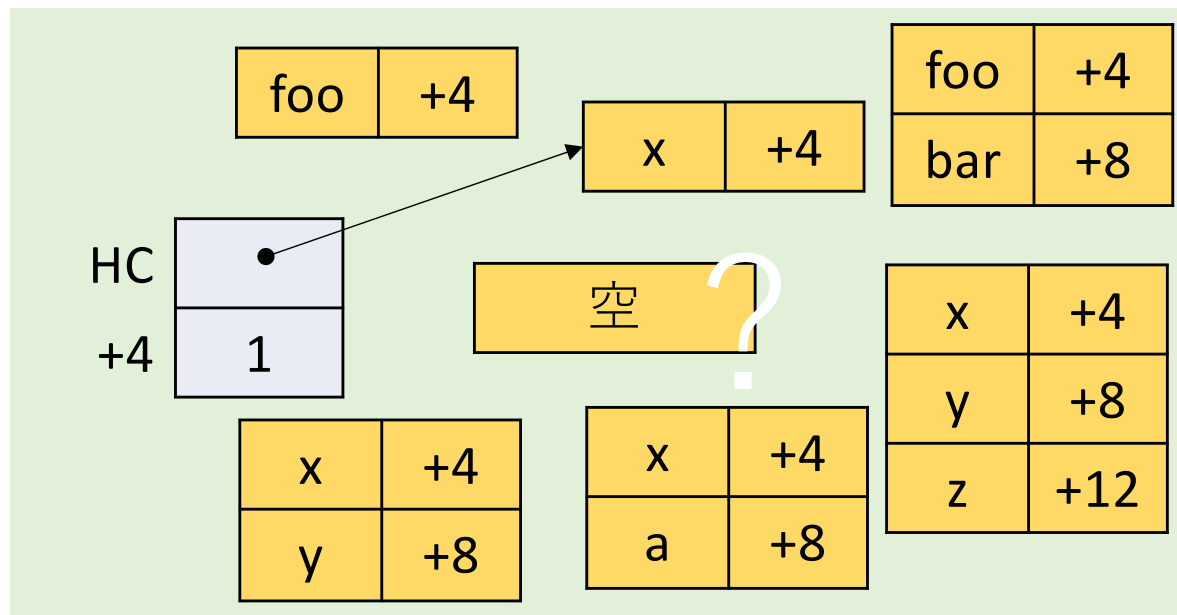
- プロパティが増えるとHCが変わる
- 共有している可能性があるのでHCはread only



hidden classを探す

- プロパティ追加時に
  既に作られたHCがあればそれを共有したい

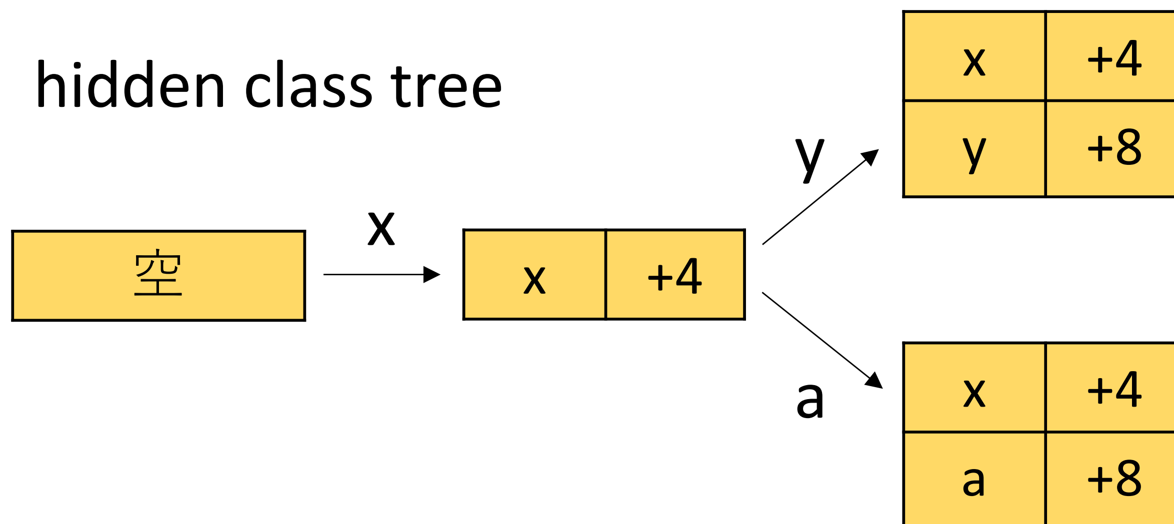
質問：どうやって次のHCを探す？




```
while(...) {
  var obj = {};
  obj.x = …;
  obj.y = …;
  …
```

hidden class transition

- HCの状態遷移図を構成する
  - 辺ラベル: 追加するプロパティ名
  - 行き先がなければ作る




```
while(...) {
  var obj = {};
  obj.x = …;
  if (...)
    obj.y = …;
  else
    obj.a = …;
}
```

ここまででできたこと

- オブジェクトに型を付ける
- メモリを節約する

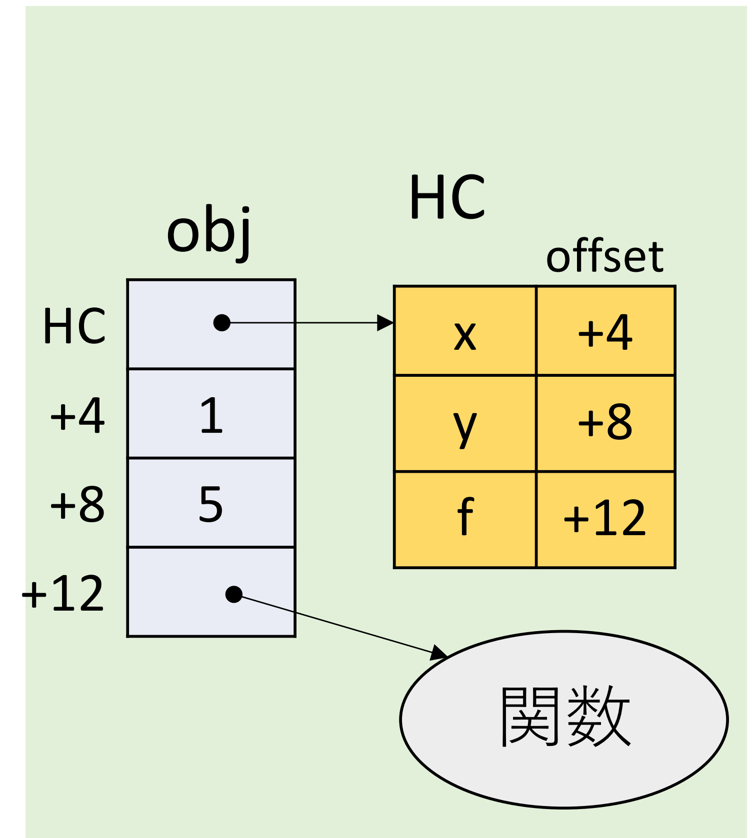
Inline caching

プロパティの検索結果をキャッシュする

プロパティのアクセス（再掲）

1. HCを検索してオフセットを得る
   - ここはハッシュ表のlookup←ここが遅い
2. オブジェクト先頭アドレス+オフセットでアクセス

```
eval(n) {
  …       // obj.prop
  if(n->type == GET_PROP){
    o = eval(n->obj);
    p = eval(n->prop);
    loc = lookup(o->HC, p);
    return *(o + loc.offset);
  }
}
```



inline caching

Deutsch, L. P. & Schiffman, A. M. :
Efficient Implementation of the Smalltalk-80 System.
POPL '84, 1984

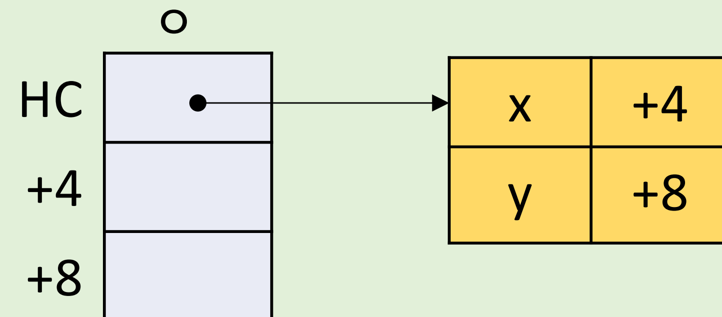
- HCの検索結果をキャッシュする
  - 各プロパティアクセス式にキャッシュ
  - インタプリタでは構文木にキャッシュ
- 仮定：同じ式では同じ型のオブジェクトを扱う

Deutsch, L. P. & Schiffman, A. M. :
Efficient Implementation of the Smalltalk-80 System.
POPL ‘84, 1984

inline caching

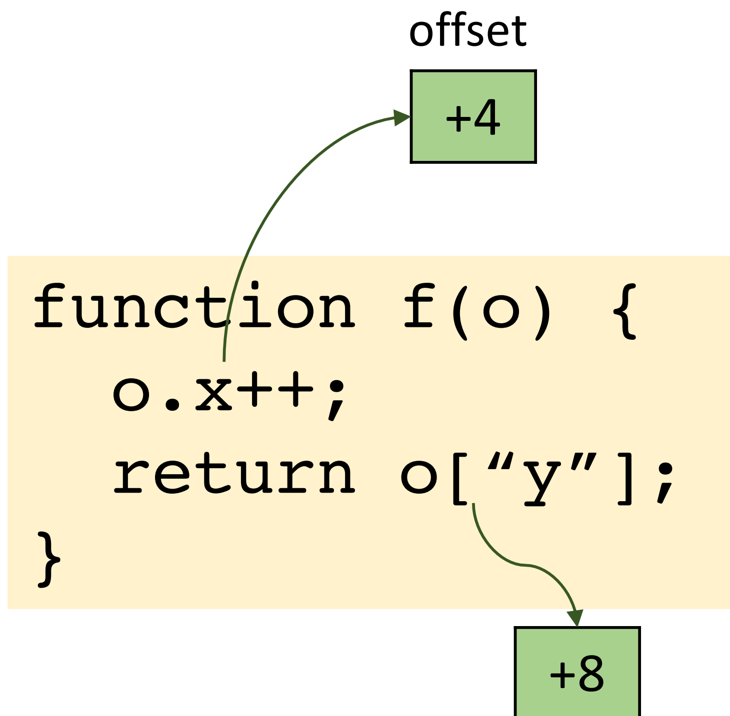
- HCの検索結果をキャッシュする
  - 各プロパティアクセス式にキャッシュ
  - インタプリタでは構文木にキャッシュ
- 仮定：同じ式では同じ型のオブジェクトを扱う

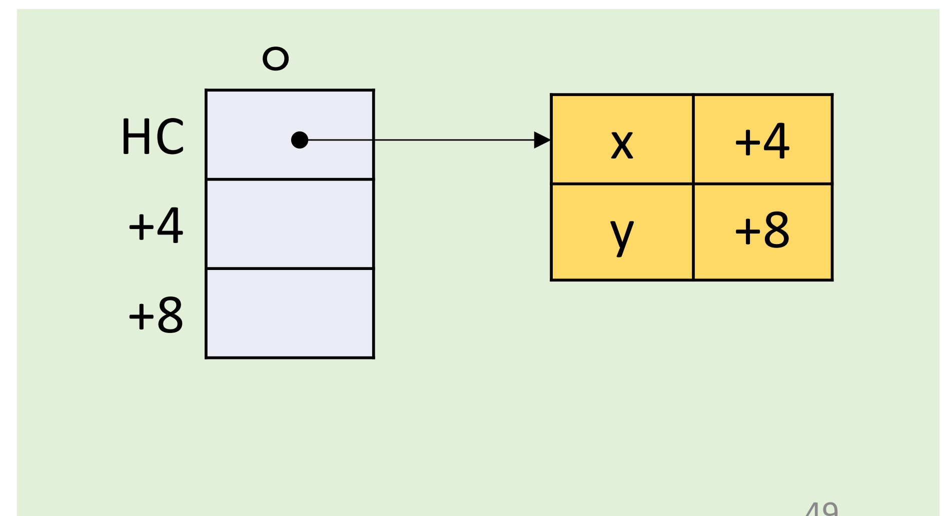
```
function f(o) {
  o.x++;
  return o["y"];
}
```



inline caching

- HCの検索結果をキャッシュする
  - 各プロパティアクセス式にキャッシュ
  - インタプリタでは構文木にキャッシュ
- 仮定：同じ式では同じ型のオブジェクトを扱う





inline cache(IC) の利用

- 2回以降はHCの検索を省略
- ICを利用

```
eval(n) {
  …          // obj.prop
  if(n->type == GET_PROP){
    o = eval(n->obj);
    p = eval(n->prop);
    if (!valid(n->IC)) {
      // 1回目だけHCから検索
      loc = lookup(o->HC, p);
      n->IC->off = loc->off;
    }
    return *(o + n->IC->off);
  }
}
```

offset

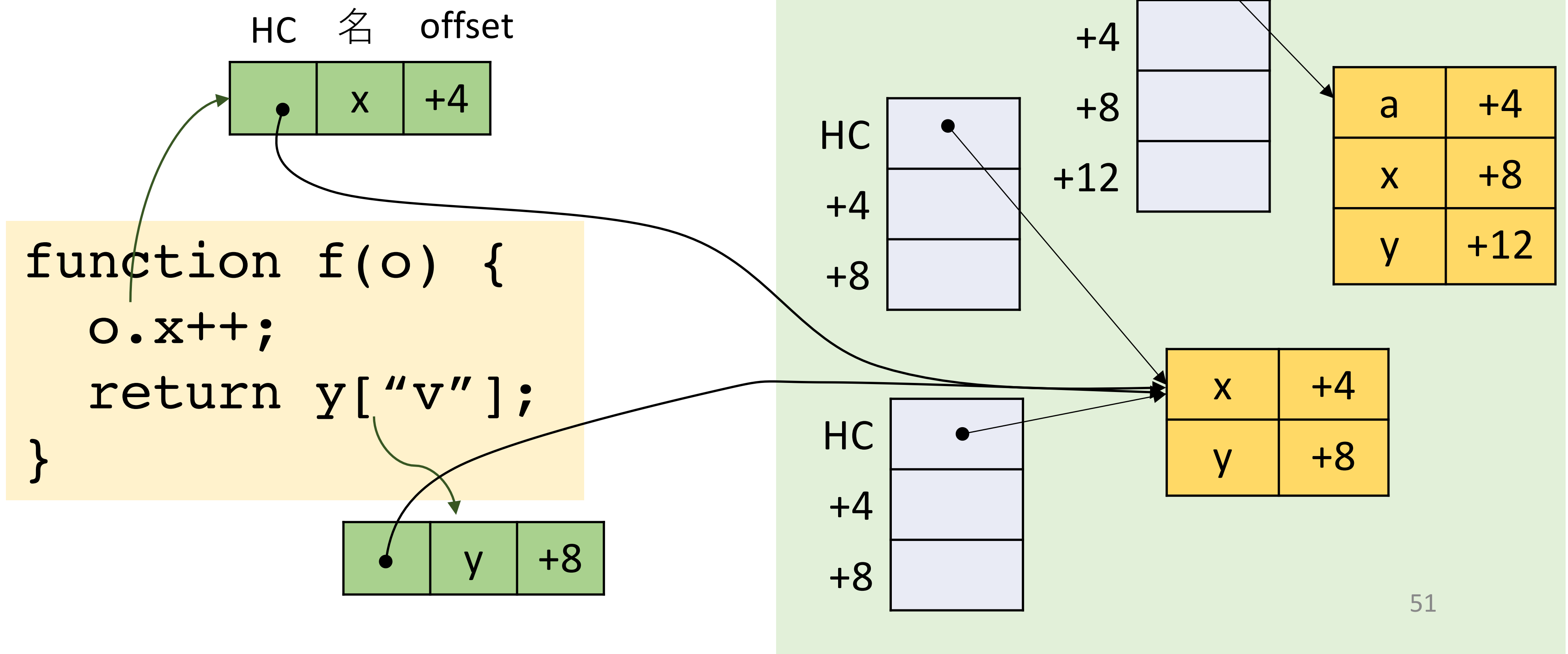
+4

```
function f(o) {
  o.x++;
  return o["y"];
}
```

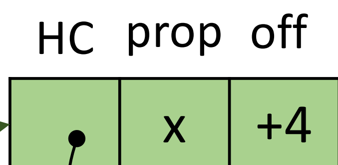
+8

ガード

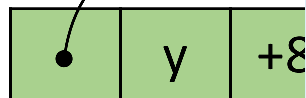
- 違う型のオブジェクトが来るかもしれない
- HCとプロパティ名を検証してから使う
  - 多くの場合
プロパティ名は検証不要



インタプリタのコード




```
function f(o) {
  o.x++;
  return o["y"];
}
```

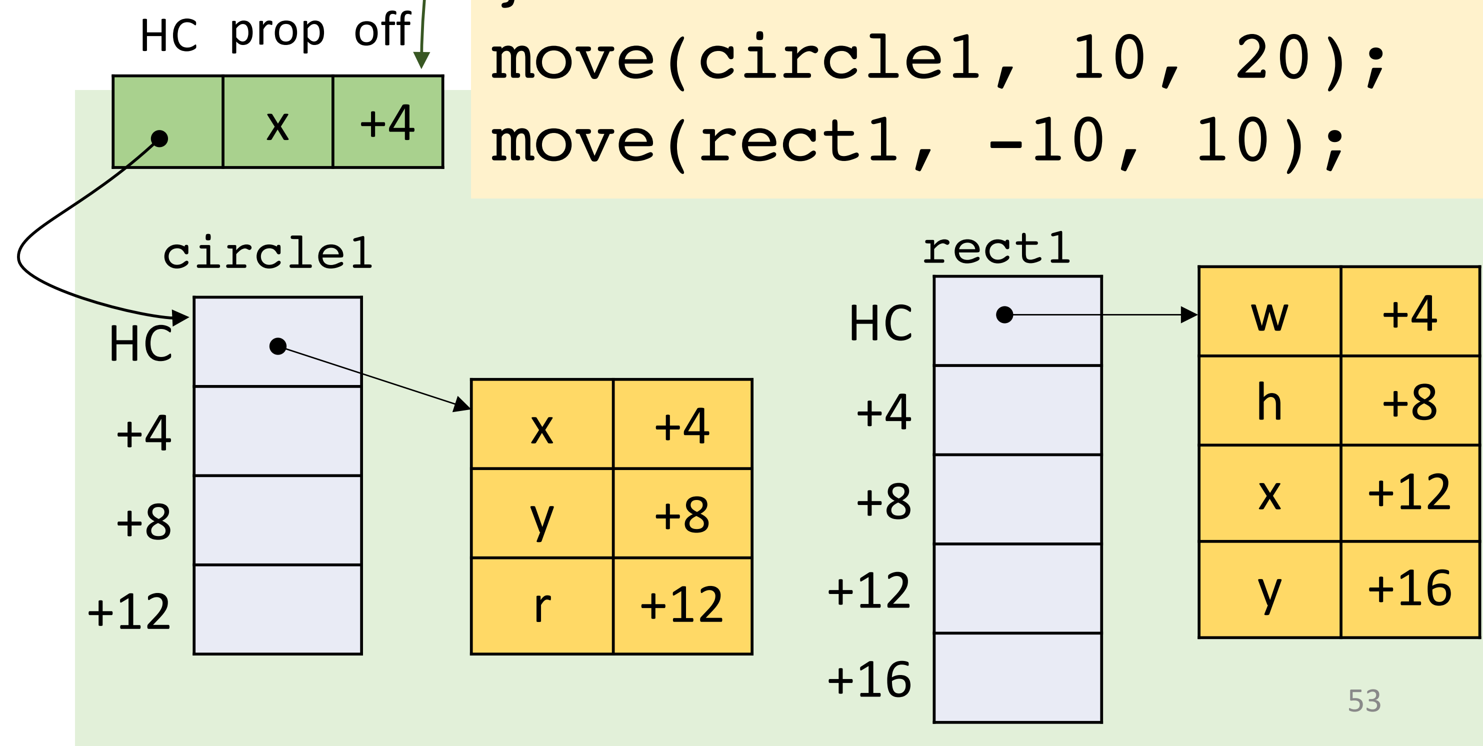



```
eval(n) {
  …        // obj.prop
  if(n->type == GET_PROP){
    o = eval(n->obj);
    p = eval(n->prop);
    if (!valid(n->IC) ||
          n->IC->HC != o->HC ||
          n->IC->prop != p ) {
      // 1回目と検証に失敗した時
      loc = lookup(o->HC, p);
      n->IC->off = loc->off;
    }
    return *(o + n->IC->off);
  }
}
```

polymorphism

- 数個の型を受け取る関数
  - 実際に結構使う
  - ICが効かない（すぐにICミスする）

```
//図形を移動させる
function move(obj,dx,dy){
  obj.x += dx;
  obj.y += dy;
}
move(circle1, 10, 20);
move(rect1, -10, 10);
```

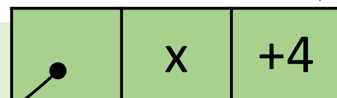


polymorphism

- 数個の型を受け取る関数
  - 実際に結構使う
  - ICが効かない（すぐにICミスする）

```
//図形を移動させる
function move(obj,dx,dy){
  obj.x += dx;
  obj.y += dy;
}
move(circle1, 10, 20);
move(rect1, -10, 10);
```

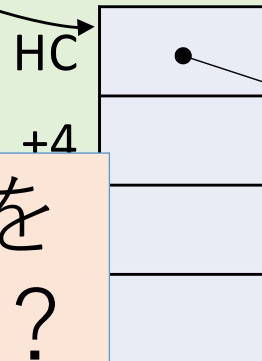
HC prop off

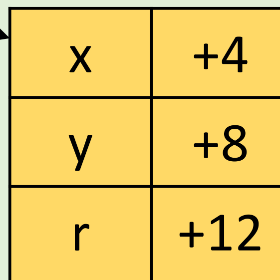


circle1

HC

+4





rect1

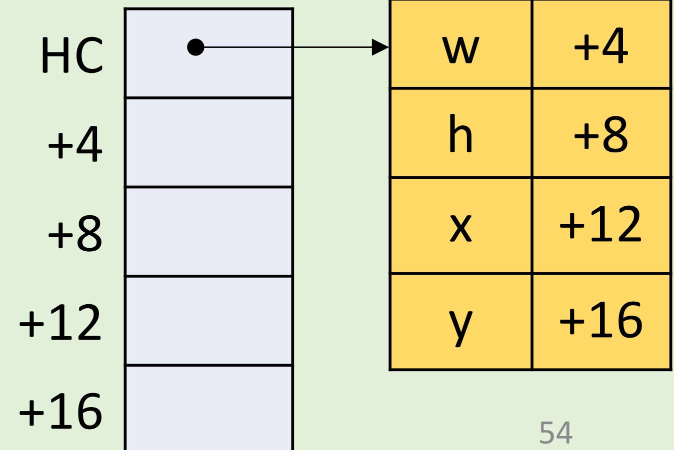
HC

+4

+8

+12

+16



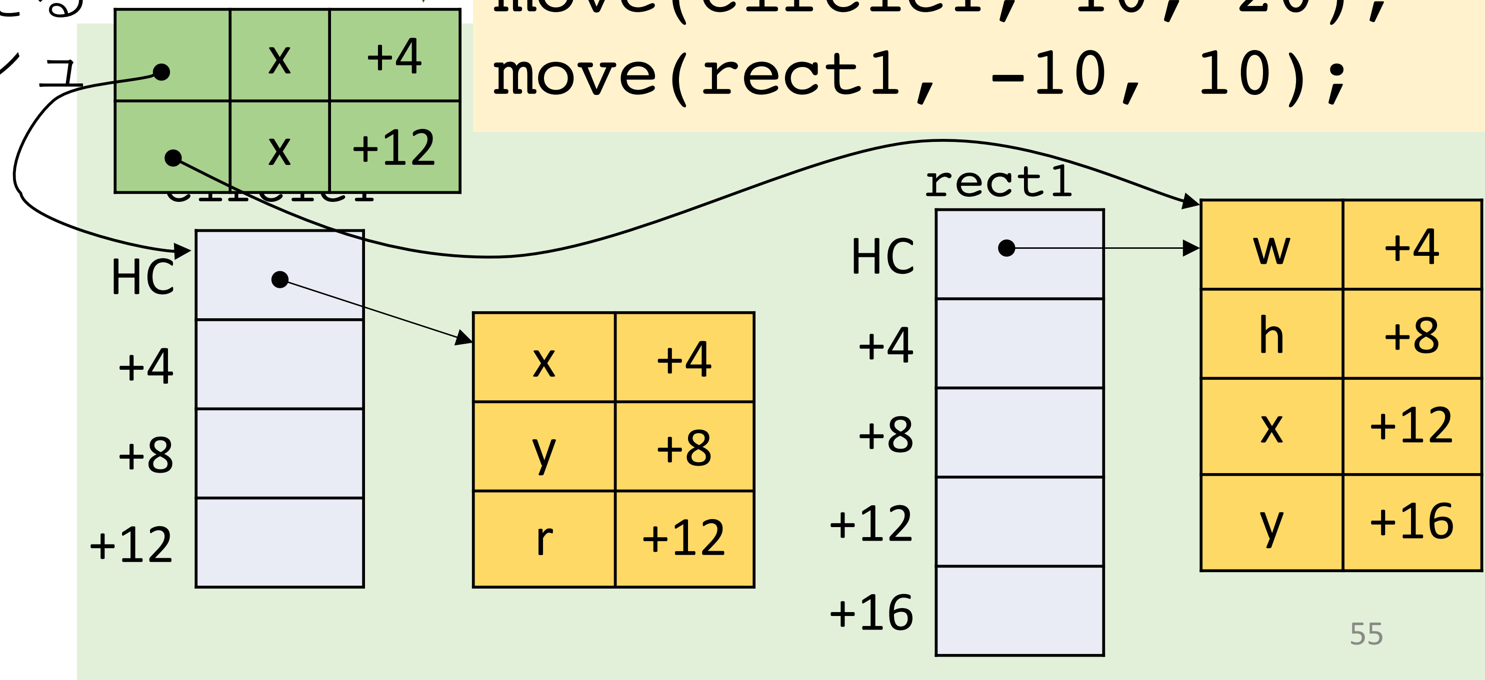
質問：ICミスを防ぐには？

Hölzle, U., Chambers, C. & Ungar, D. :
Optimizing Dynamically-Typed Object-Oriented Languages With Polymorphic Inline Caches.
ECOOP'91, 1991

polymorphic inline cache

- 複数キャッシュする
  - 仮定：使われる型は限られている
- エントリ数の上限
  - 上限を超えるとキャッシュミス

```
//図形を移動させる
function move(obj,dx,dy){
  obj.x += dx;
  obj.y += dy;
}
move(circle1, 10, 20);
move(rect1, -10, 10);
```



実験

```
function move(obj,dx,dy){
  obj.x += dx;
  obj.y += dy;
}
```

moveに渡すオブジェクトの型の数(K)と実行時間

- node.js v10.19.0
- Linux Ubuntu 12.04, Intel core i9-10920X

```
a[0] = {“x”:0, “y”:0, “z0”:0};
a[1] = {“x”:0, “y”:0, “z1”:0};
…
a[9] = {“x”:0, “y”:0, “z9”:0};
var start = performance.now();
for (i = 0; i < 10000000; i++)
  move(a[i % K], 0, 0); // K種類渡す
var end = performance.now();
```

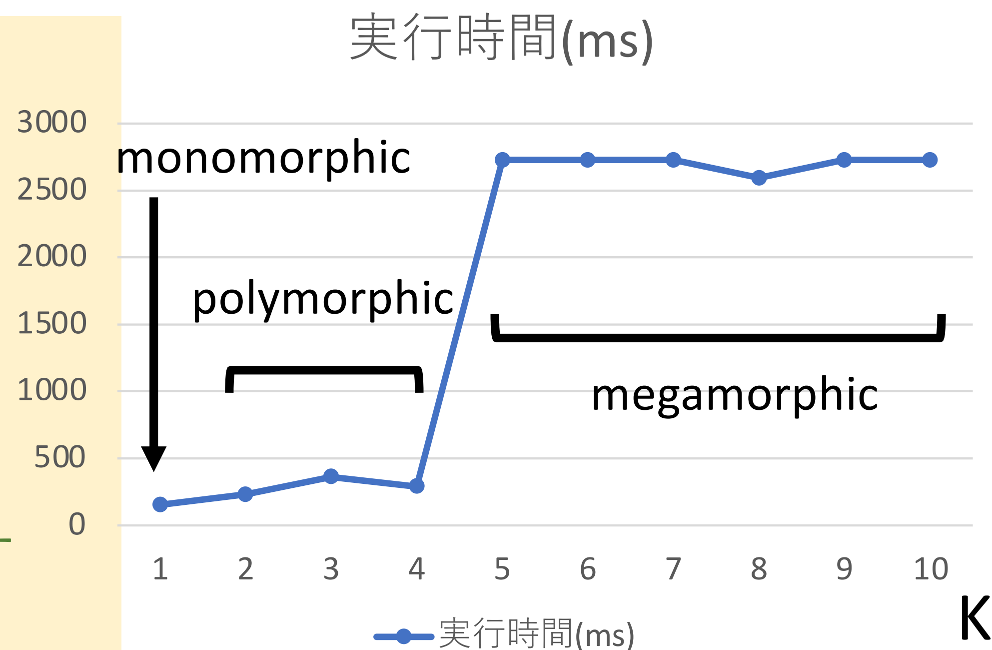
実験

```
function move(obj,dx,dy){
  obj.x += dx;
  obj.y += dy;
}
```

moveに渡すオブジェクトの型の数(K)と実行時間

- node.js v10.19.0
- Linux Ubuntu 12.04, Intel core i9-10920X

```
a[0] = {“x”:0, “y”:0, “z0”:0};
a[1] = {“x”:0, “y”:0, “z1”:0};
…
a[9] = {“x”:0, “y”:0, “z9”:0};
var start = performance.now();
for (i = 0; i < 10000000; i++)
  move(a[i % K], 0, 0); // K種類渡す
var end = performance.now();
```



実験2

```
function move(obj,dx,dy){
  obj.x += dx;
  obj.y += dy;
}
```

同じプロパティを与える

```
a[0] = {“x”:0, “y”:0, “z0”:0};
a[1] = {“x”:0, “y”:0, “z1”:0};
…
a[9] = {“x”:0, “y”:0, “z9”:0};
for (i = 0; i < 10; i++)
  a[i].z0 = a[i].z1 = … = a[i].z9 = 0;
var start = performance.now();
for (i = 0; i < 10000000; i++)
  move(a[i % K], 0, 0);
var end = performance.now();
```

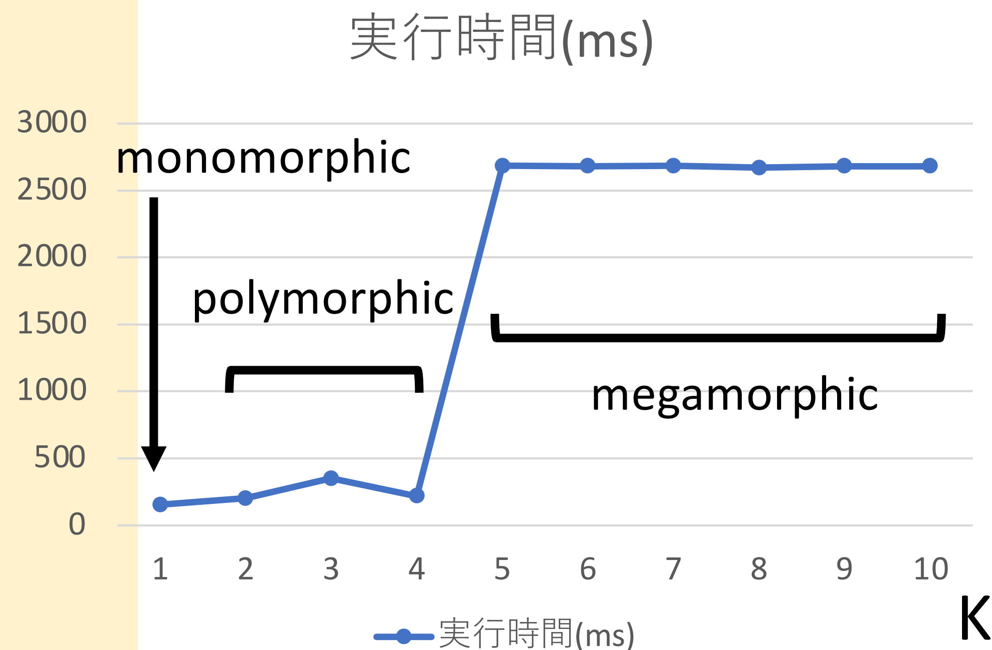
質問：結果はどうなる？

実験2

```
function move(obj,dx,dy){
  obj.x += dx;
  obj.y += dy;
}
```

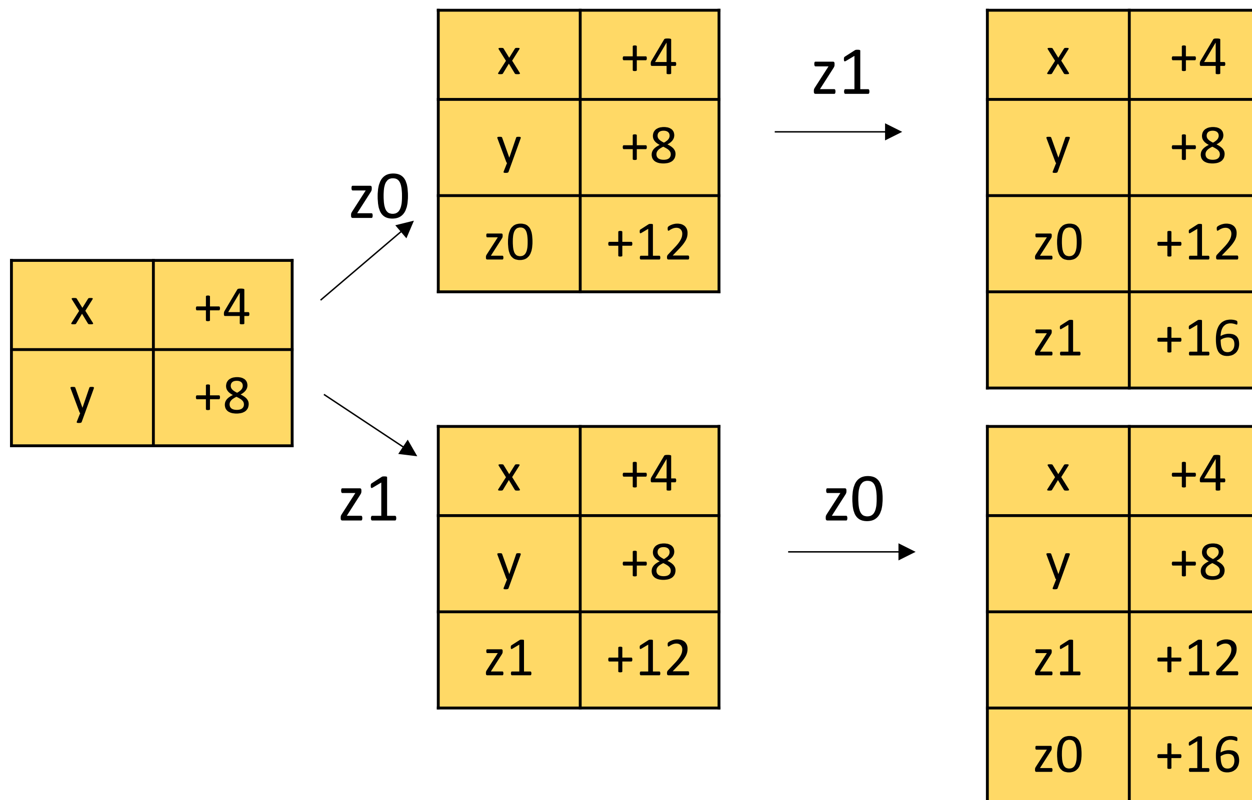
結果は変わらない

```
a[0] = {"x":0, "y":0, "z0":0};
a[1] = {"x":0, "y":0, "z1":0};
…
a[9] = {"x":0, "y":0, "z9":0};
for (i = 0; i < 10; i++)
  a[i].z0 = a[i].z1 = … = a[i].z9 = 0;
var start = performance.now();
for (i = 0; i < 10000000; i++)
  move(a[i % K], 0, 0);
var end = performance.now();
```



タネ明かし

- プロパティを追加する順序が異なる
  => レイアウトが異なる



ここまでできたこと

- 検索結果のキャッシュ
- HCの検索コストが長期的には0になる

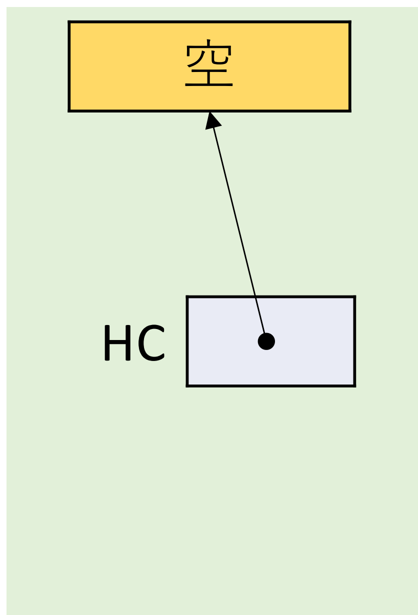
- どうせHCの検索コストがないのなら…
  - HCをコンパクトな表現にする
  - HCにもっと情報を載せる
    - プロパティの型とか

advanced topics

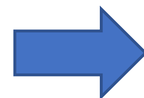
プロパティの追加

質問：実装するときに困ることは？

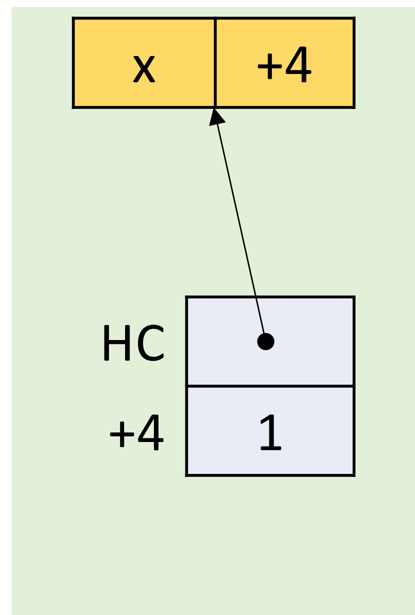
hidden class 0



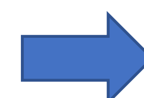
obj.x=1



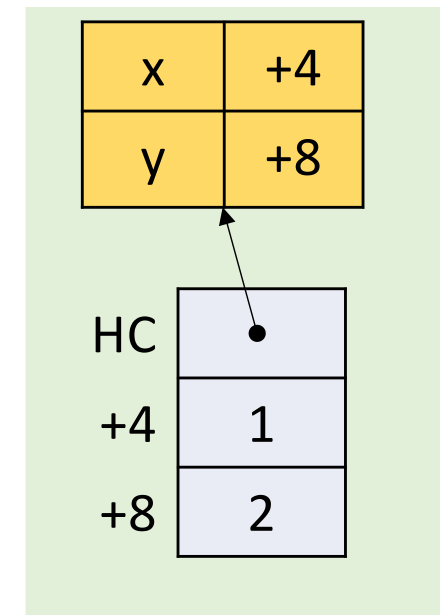
hidden class 1



obj.y=2

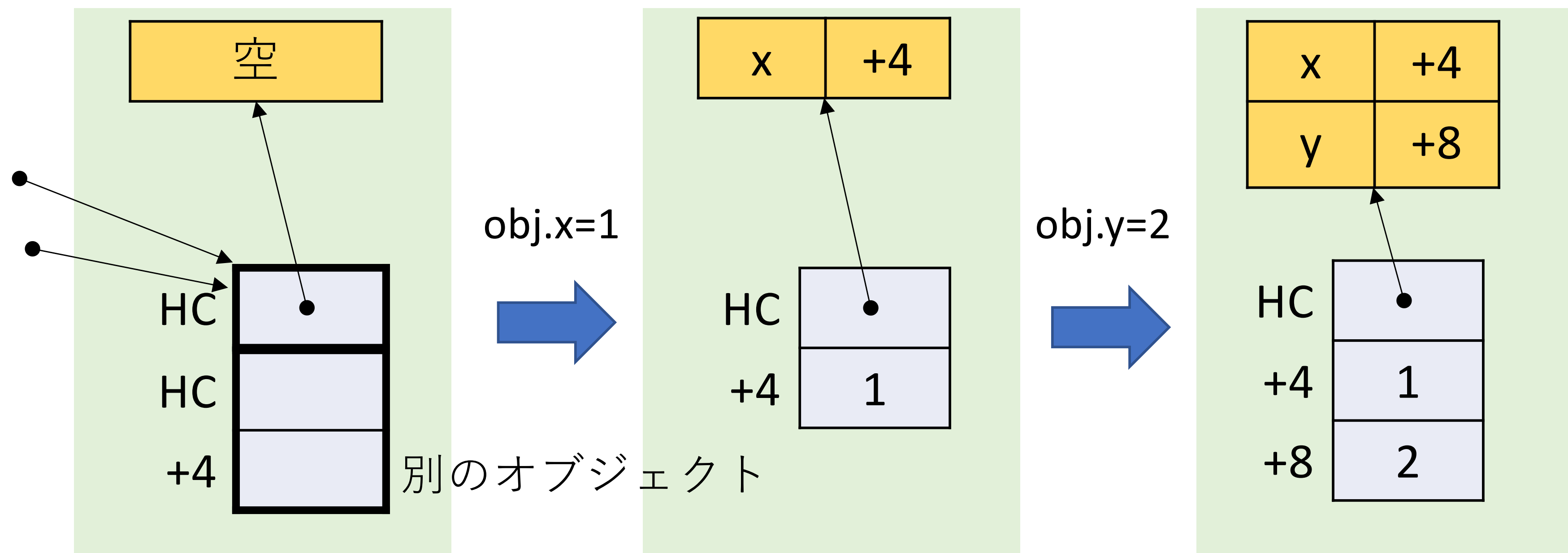


hidden class 2



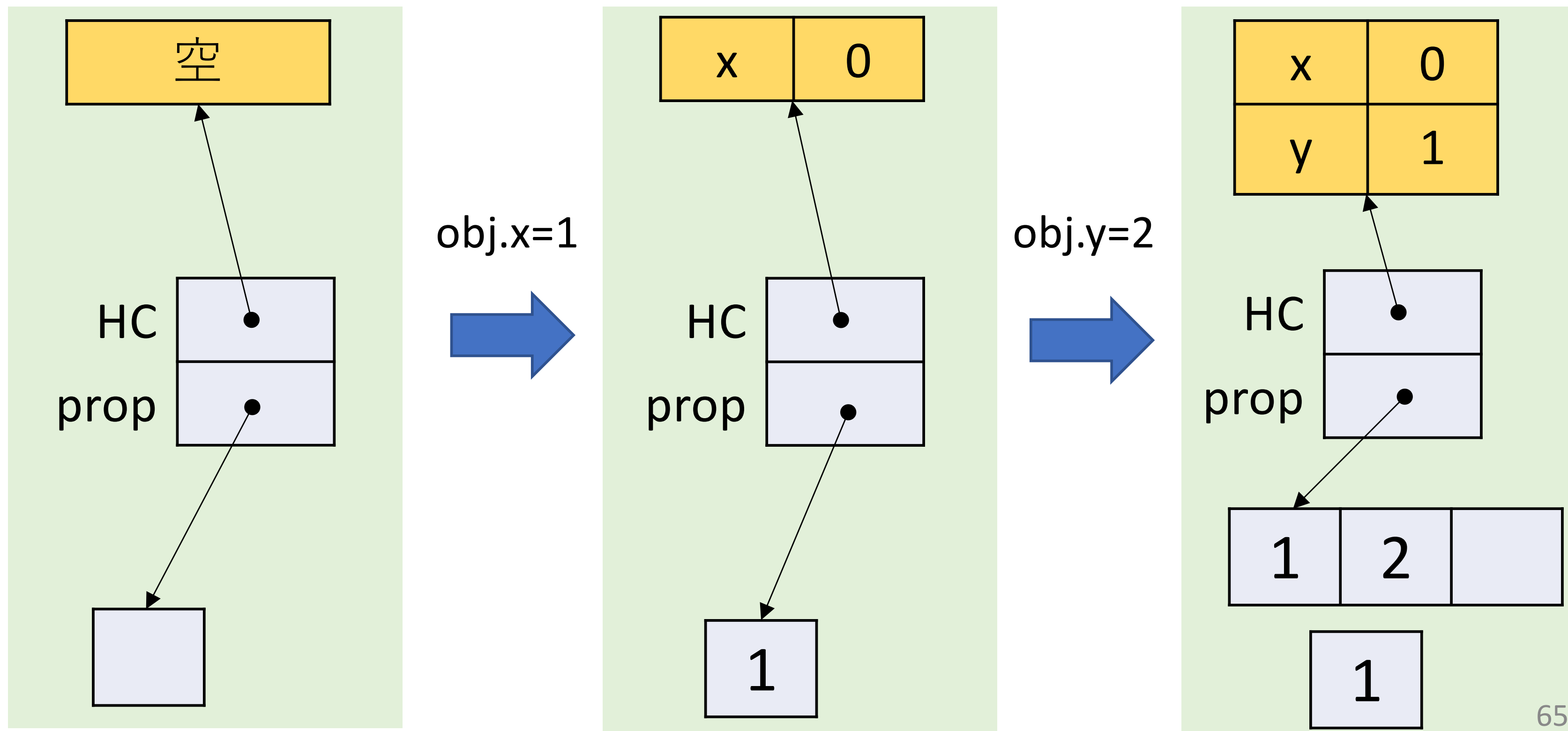
プロパティの追加

- オブジェクトをその場で拡張できない
  - 直後に別のオブジェクトが割り当てられている
- 移動させるのも難しい
  - どこから指されているか分からない



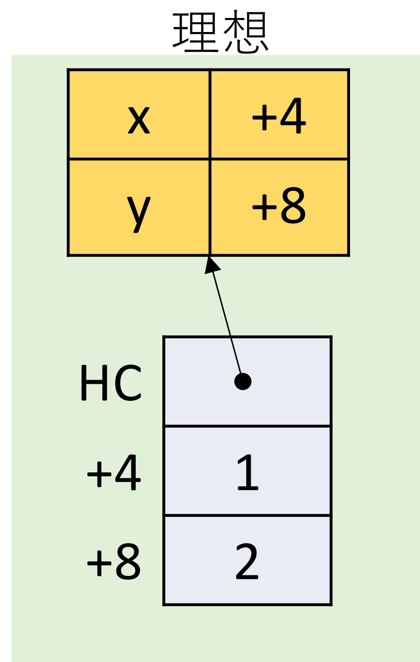
naïveな実装：プロパティ配列の分離

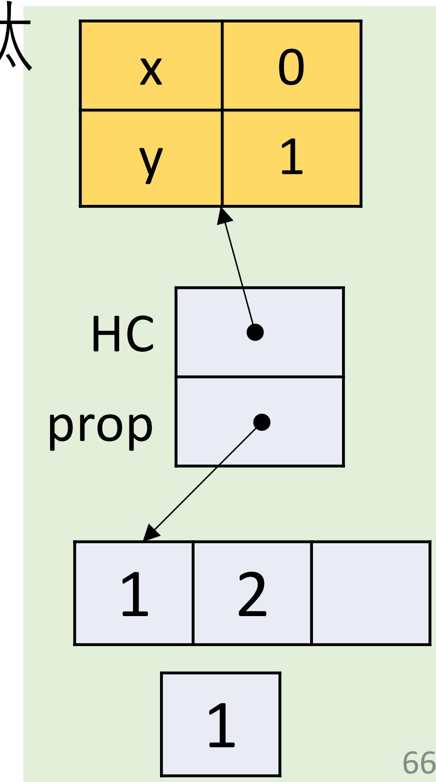
- プロパティの配列をオブジェクト外部に用意
- 必要に応じてreallocate



naïveな実装の問題

- 遅い
  - reallocateが多い
  - プロパティアクセスが間接アクセスになる
- 余分にメモリを割り当てるのは無駄

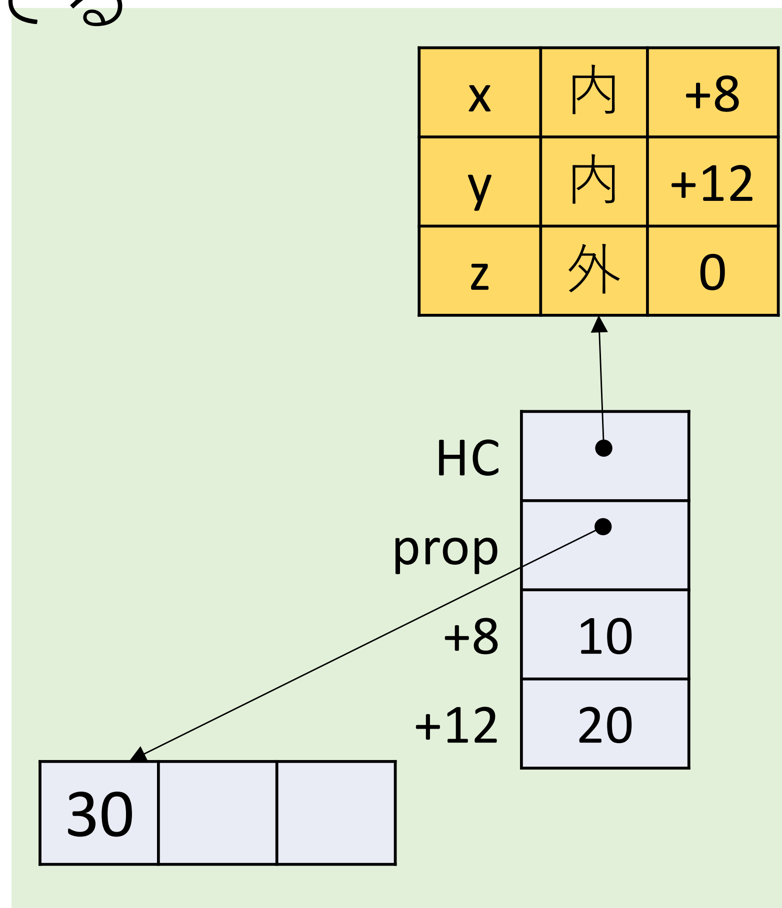




advancedな実装

- オブジェクトを作るときに、いくつかオブジェクト内にスロットを割り当てる
- 溢れたら外部に割り当てる

```
function Point(x,y) {
 this.x = x;
 this.y = y;
}
var p = new Point(10,20);
p.z = 30;
```



in-objectプロパティの数は？

質問：in-objectプロパティの数を決める
良い戦略は？

in-objectプロパティの数は？

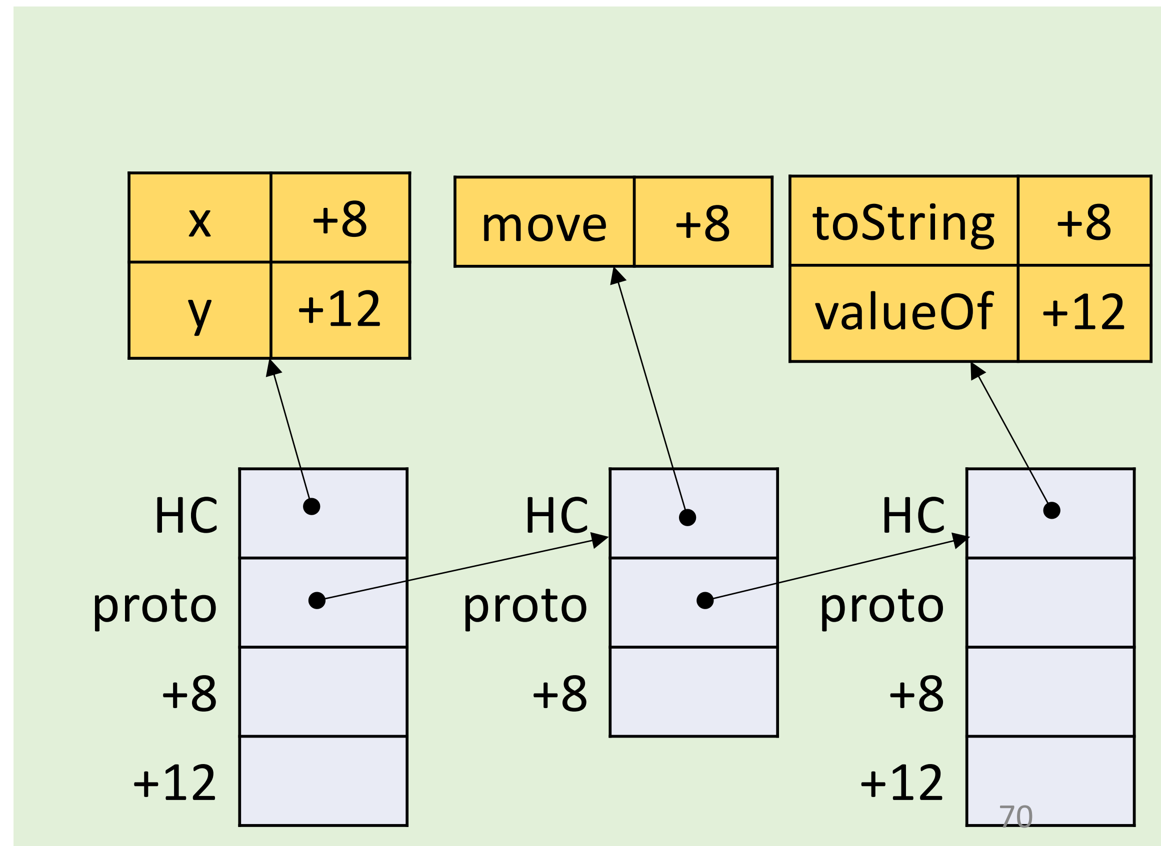
- ハードコードされた定数
- オブジェクトを追跡調査して最終的なプロパティ数を調べる
  - コンストラクタ関数ごと
    slack tracking (https://v8.dev/blog/slack-tracking)
  - オブジェクトを作る式ごと
    eJS (MoreVMs ’2021)

プロトタイプ

- 個々のオブジェクトがプロトタイプを持つ
- プロパティを読み出すとき、プロトタイプも検索する

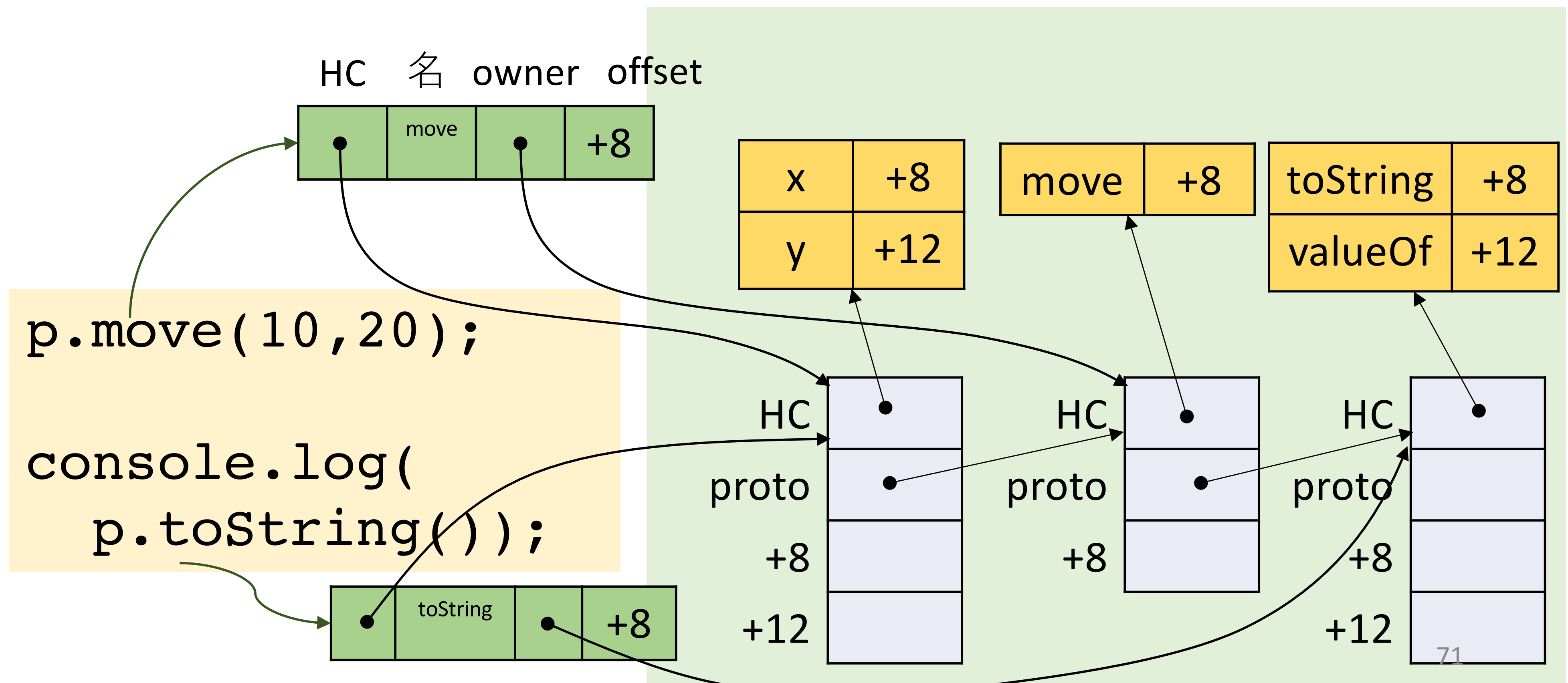
```
p.move(10,20);

console.log(
  p.toString());
```



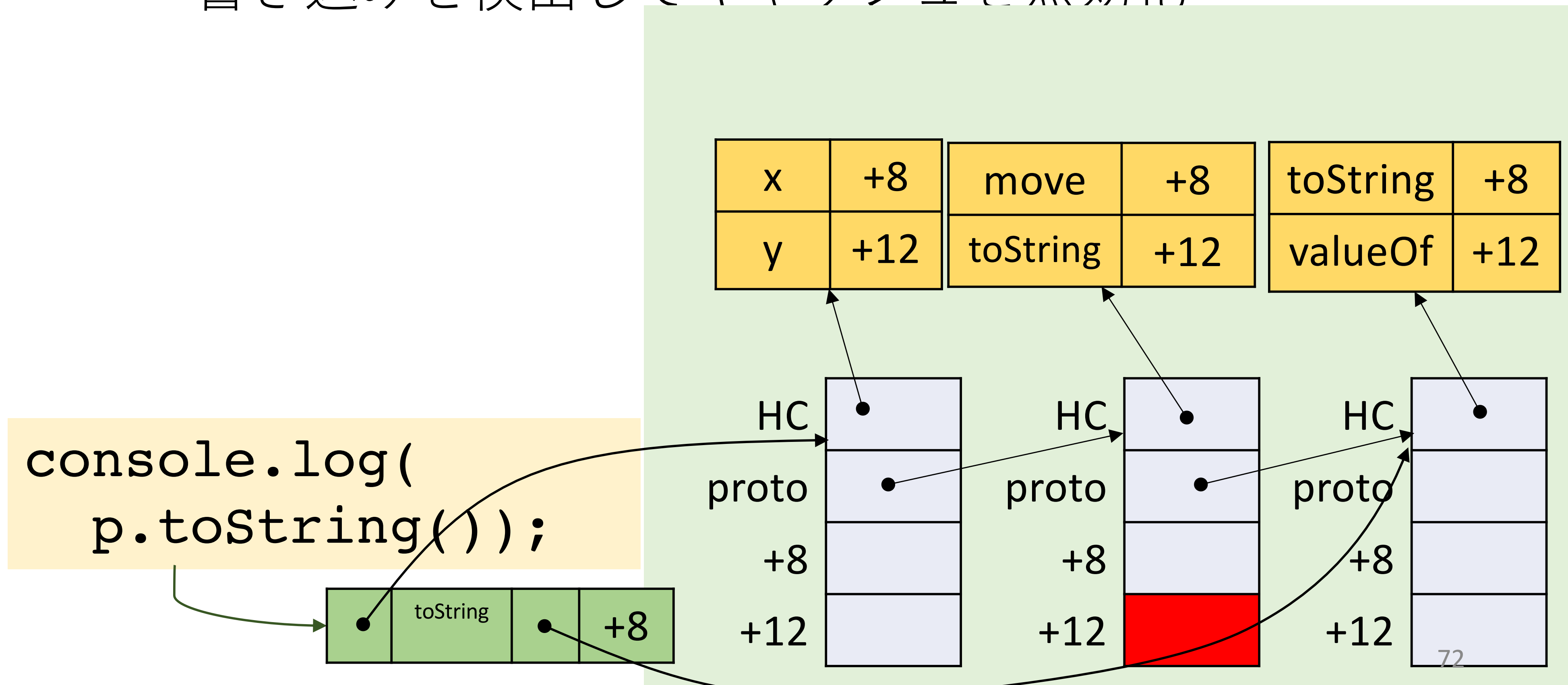
inline caching

- オーナオブジェクトも記録する



プロトタイプへの書き込み

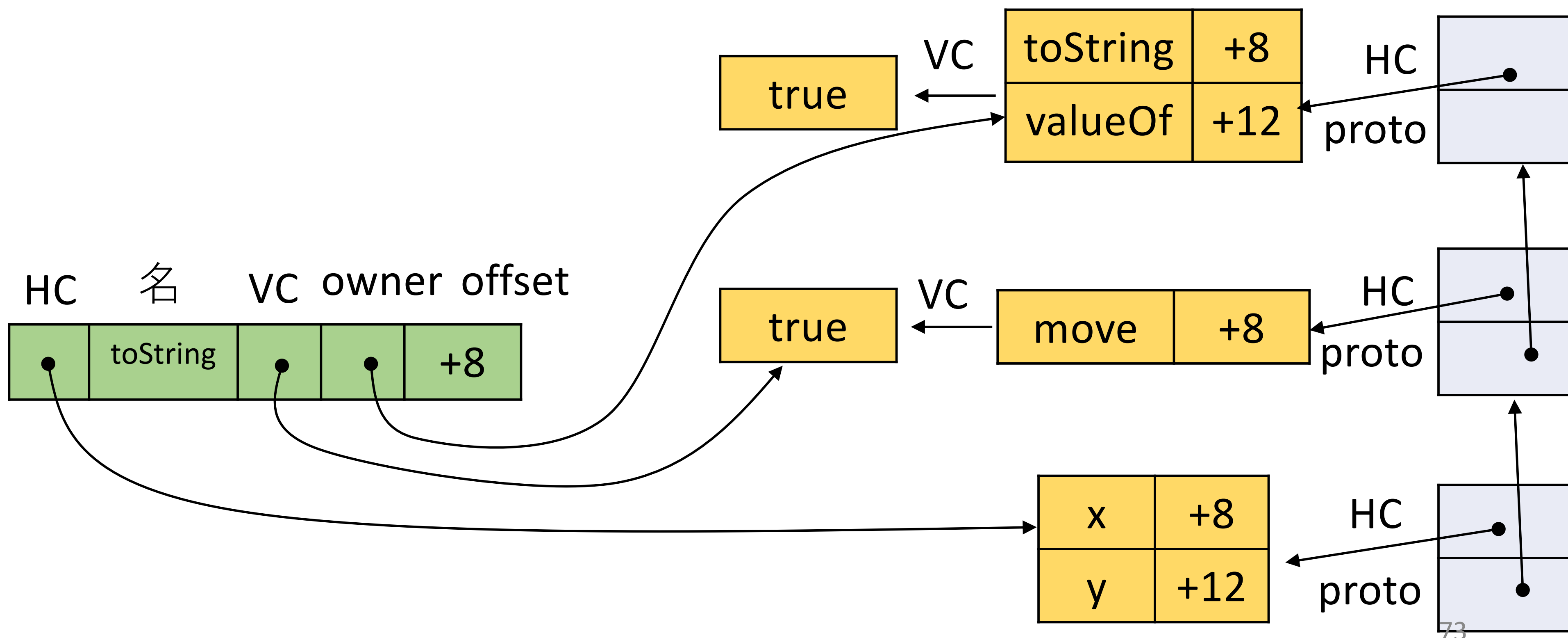
- より“近い”オブジェクトに同じ名前のプロパティを追加すると隠蔽される
  - 書き込みを検出してキャッシュを無効化



validity cell (VC)

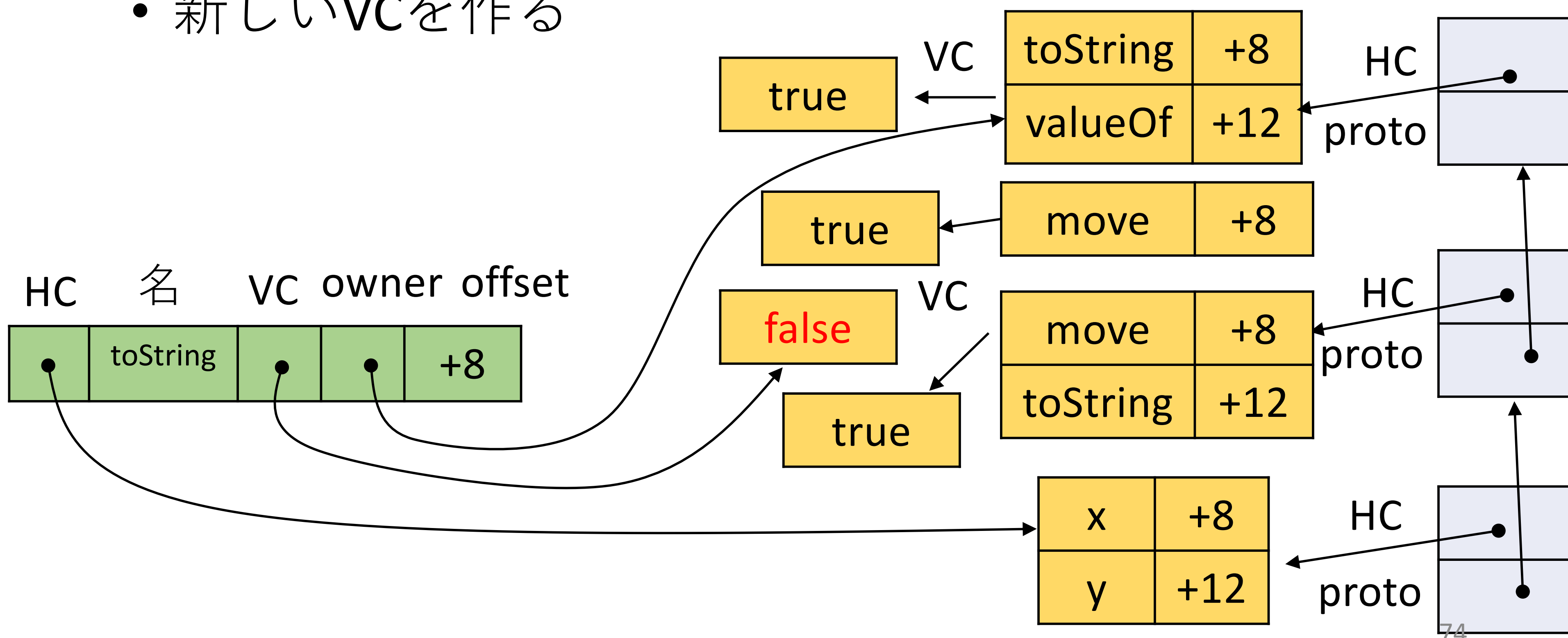
https://mathiasbynens.be/notes/prototypes

- キャッシュされたHCが最新であることを示すフラグのオブジェクト
- キャッシュを使う前にVCも検証する



プロパティの追加

- VCを無効化する
  - キャッシュされている（かもしれない）VCをfalseにする
  - 新しいVCを作る



まとめ

- 動的言語を静的にコンパイルしてもなかなか性能が出ない
  - プロパティ検索、メソッド検索
  - 型ディスパッチ
- 実行時に分かる情報を使うと高速化できる
  - hidden class: オブジェクトに型を付ける
  - inline caching: 検索結果をキャッシュする
    - 静的言語と違ってキャッシュが利用できるかの検証（ガード）とバックアップの仕組みは必要

話さなかったこと

- 配列の効率化
  - Storage storategy
    C. F. Bolz, L. Diekmann, and L. Tratt. :
    Storage Strategies for Collections in Dynamically Typed Languages. OOPSLA ‘13, 2013
  - pre-transitioning
    Clifford, D., Payer, H., Stanton, M. & Titzer, B. L.: **Memento Mori: Dynamic Allocation-site-based Optimizations.** ISMM ’15, 2015
- プリミティブデータの効率化
  - プロパティに型をつける
    https://v8.dev/blog/react-cliff
  - Nan-boxing
    JavaScript coreのJSCJSValue.hのコメント
    https://github.com/WebKit/WebKit/blob/main/Source/JavaScriptCore/runtime/JSCJSValue.h