

Stack Allocation y otros Monstruos

Pedro Palao Gostanza
Seismo Technologies

2024-10-30



Definición

Stack Allocation es una optimización por la que un objeto creado con `new C(...)` se aloja realmente en la pila

Definición

Stack Allocation es una **optimización** por la que un objeto creado con `new C(...)` se aloja realmente en la pila

Toma de contacto

Donald Knuth / Tony Hoare

Premature optimization is the root of all evil.

Toma de contacto

Donald Knuth / Tony Hoare

We should forget about small efficiencies, say about 97% of the time:
Premature optimization is the root of all evil. Yet we should not pass up
our opportunities in that critical 3%

Toma de contacto

Diseñar para

- Corrección
- Mantenimiento
- Seguridad
- Testeo
- ...

Toma de contacto

Diseñar para

- Corrección
- Mantenimiento
- Seguridad
- Testeo
- ...
- **Rendimiento**

Conceptos

Los 3 conceptos fundamentales

- Escape Analysis
- Stack Allocation
- Scalar Replacement

Conceptos

Ejemplo: hash heap

```
1 long hash(byte[] d) {  
2     var h = new CRC32();  
3     h.update(d);  
4     return h.getValue();  
5 }
```

Conceptos

Ejemplo: hash heap

```
1 long hash(byte[] d) {  
2     var h = new CRC32();  
3     h.update(d);  
4     return h.getValue();  
5 }
```

Stack

Conceptos

Ejemplo: hash heap

```
1 long hash(byte[] d) {  
2     var h = new CRC32();  
3     h.update(d);  
4     return h.getValue();  
5 }
```

Stack

Conceptos

Ejemplo: hash heap

```
1 long hash(byte[] d) {  
2     var h = new CRC32();  
3     h.update(d);  
4     return h.getValue();  
5 }
```

Stack

? | d | h

Conceptos

Ejemplo: hash heap

```
1 long hash(byte[] d) {  
2     var h = new CRC32();  
3     h.update(d);  
4     return h.getValue();  
5 }
```

Stack

? | d | h

Heap

Conceptos

Ejemplo: hash heap

```
1 long hash(byte[] d) {  
2     var h = new CRC32();  
3     h.update(d);  
4     return h.getValue();  
5 }
```

Stack

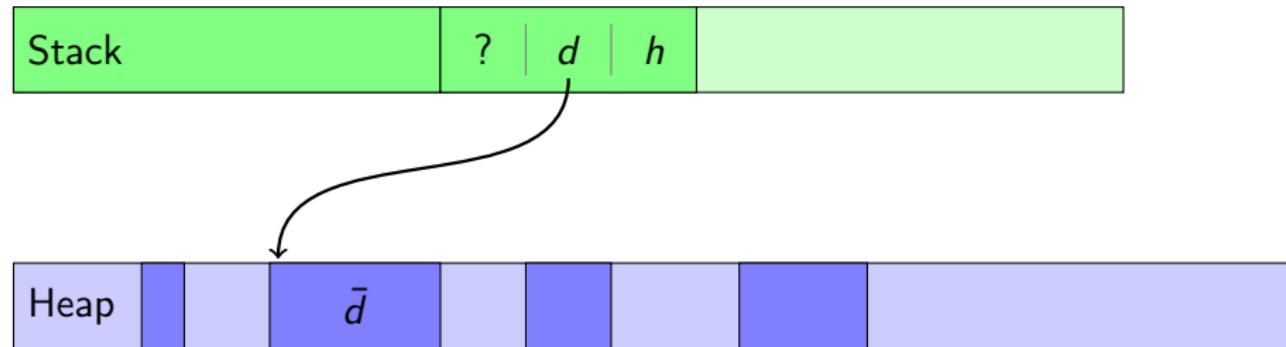
? | d | h

Heap

Conceptos

Ejemplo: hash heap

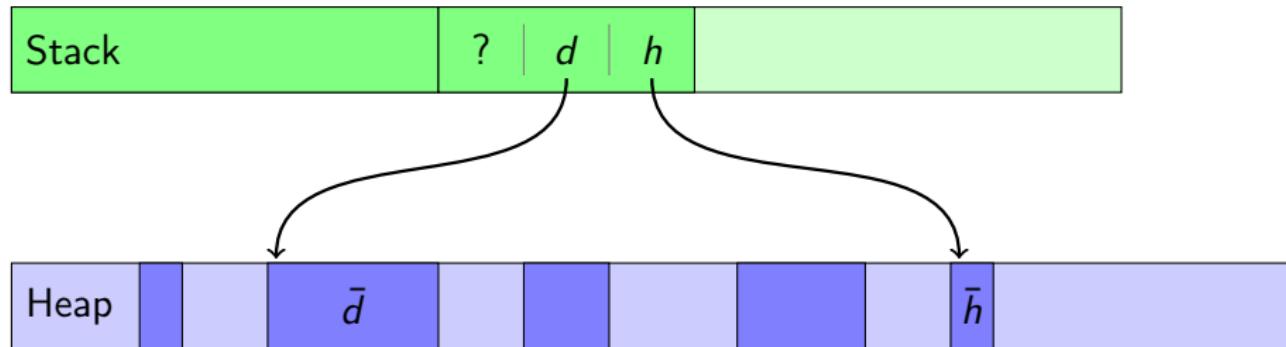
```
1 long hash(byte[] d) {  
2     var h = new CRC32();  
3     h.update(d);  
4     return h.getValue();  
5 }
```



Conceptos

Ejemplo: hash heap

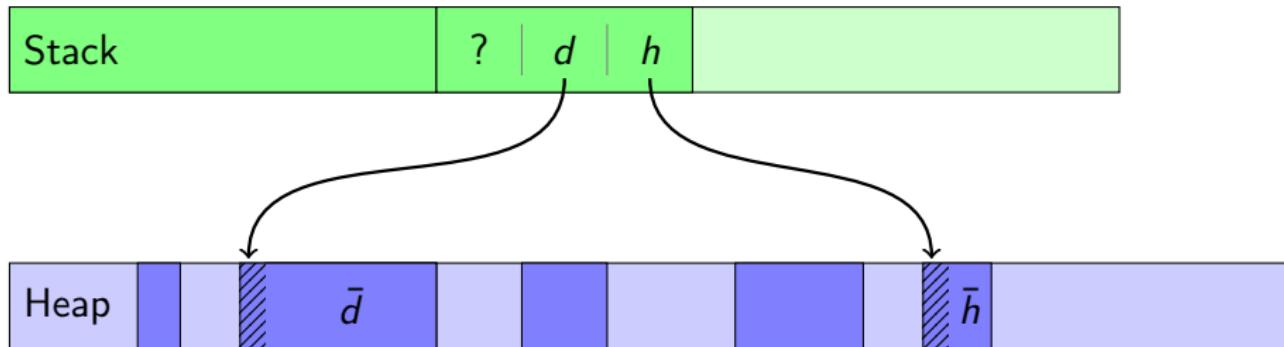
```
1 long hash(byte[] d) {  
2     var h = new CRC32();  
3     h.update(d);  
4     return h.getValue();  
5 }
```



Conceptos

Ejemplo: hash heap

```
1 long hash(byte[] d) {  
2     var h = new CRC32();  
3     h.update(d);  
4     return h.getValue();  
5 }
```

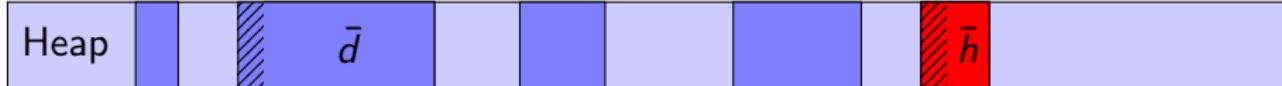


Conceptos

Ejemplo: hash heap

```
1 long hash(byte[] d) {  
2     var h = new CRC32();  
3     h.update(d);  
4     return h.getValue();  
5 }
```

Stack



Escape Analysis

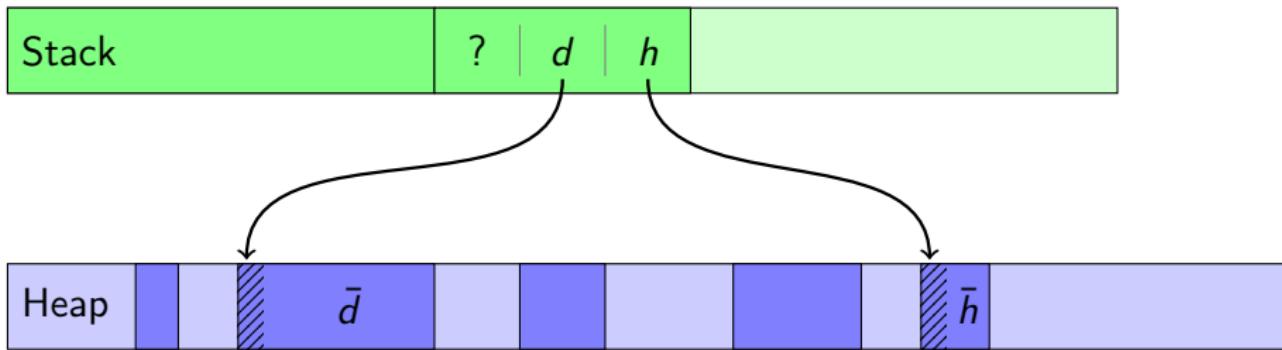
- Si un objeto solo vive mientras está activa la llamada al método, entonces se puede alojar en la pila
- Para confirmar que no sobrevive:
 - ▶ No se devuelve,
 - ▶ No se guarda en otro objeto
 - ▶ Ni en una variable global
 - ▶ ...

Esto es **Escape Analysis Pa' Trás**

Conceptos

Ejemplo: hash stack

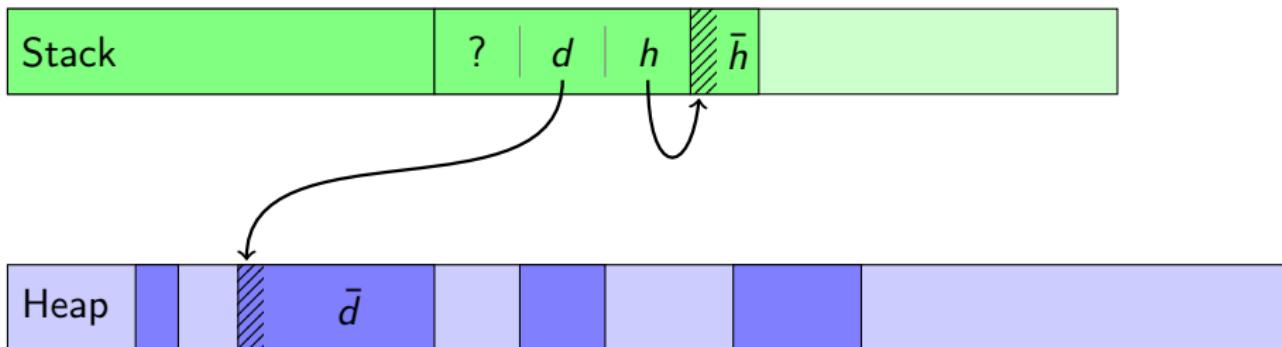
```
1 long hash(byte[] d) {  
2     var h = new CRC32();  
3     h.update(d);  
4     return h.getValue();  
5 }
```



Conceptos

Ejemplo: hash stack

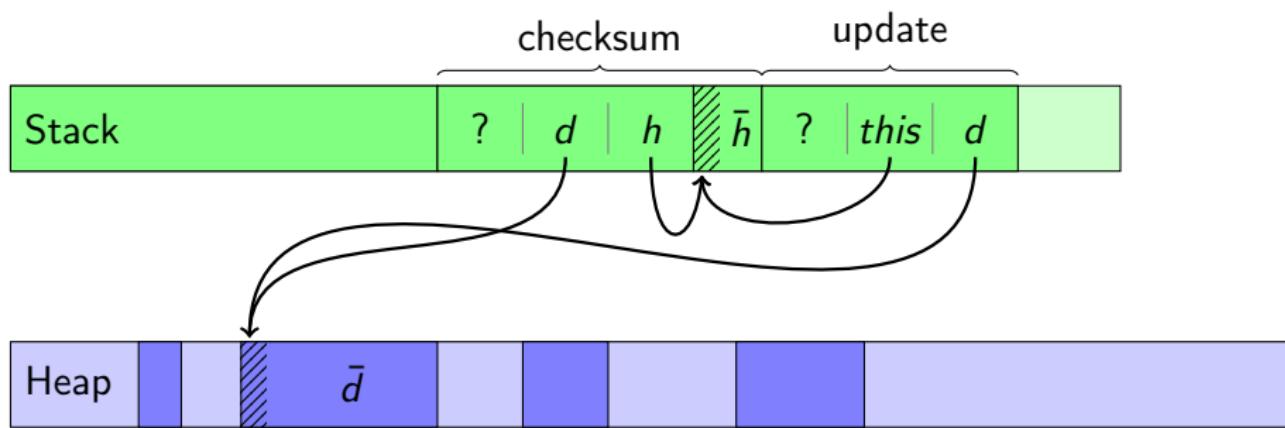
```
1 long hash(byte[] d) {  
2     var h = new CRC32();  
3     h.update(d);  
4     return h.getValue();  
5 }
```



Conceptos

Ejemplo: hash stack

```
1 long hash(byte[] d) {  
2     var h = new CRC32();  
3     h.update(d);  
4     return h.getValue();  
5 }
```



Implicaciones de colocar objetos completos en la pila

- Afecta a detalles importantes en GC:
 - ▶ Write barriers no tendría efectos sobre los objetos en la pila
 - ▶ Busqueda de raíces en la pila
 - ▶ No habría que hacer marcado ni barrido de los objetos en la pila
- Puede verse afectado por desoptimizaciones
- Mejora el rendimiento
 - ▶ Es más sencillo colocar el objeto en la pila
 - ▶ No se genera basura
 - ▶ Hay efectos indirectos: más localidad en los datos
- Permite sincronizaciones
- Permite *escape Pa'Lante*

Conceptos

Monstruo 1: Inlining: la madre de todas las optimizaciones

Reemplazar una llamada a un método por su código

Código relevante de CRC32

```
1  public interface Checksum {  
2      default void update(byte[] b) {update(b,0,b.length);}  
3  }  
4  public class CRC32 implements Checksum {  
5      private int crc = 0;  
6  
7      public long getValue() {return (long)crc & 0xffffffffL;}  
8  
9      public void update(byte[] b, int off, int len) {  
10         crc = updateBytes(crc, b, off, len);  
11     }  
12     static int updateBytes(int crc,  
13             byte[] b, int off, int len) {  
14         return updateBytes0(crc, b, off, len);  
15     }  
16     @IntrinsicCandidate  
17     static native int updateBytes0(int crc,  
18             byte[] b, int off, int len);  
19 }
```

Conceptos

Ejecución reiterada del método hash

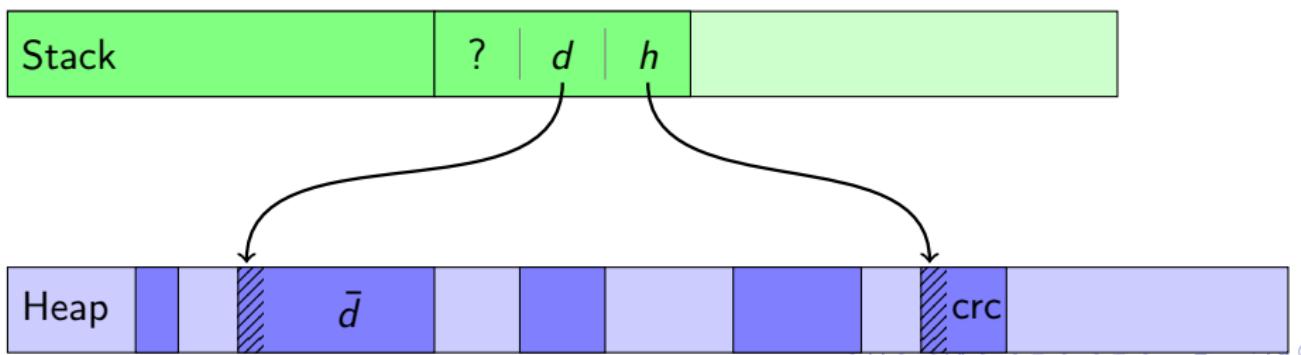
```
1 -XX:+UnlockDiagnosticVMOptions
2 -XX:+PrintCompilation
3 -XX:+PrintInlining

1 ::hash (22 bytes)  made not entrant
2 @ 4 CRC32::<init> (5 bytes)  inline (hot)
3   @ 1 Object::<init> (1 bytes)  inline (hot)
4 @ 10 Checksum::update (11 bytes)  inline (hot)
5   @ 5 CRC32::update (38 bytes)  inline (hot)
6     @ 31 CRC32::updateBytes (14 bytes)  inline (hot)
7       @ 10 CRC32::updateBytes0 (0 bytes)  (intrinsic)
8 @ 16 CRC32::getValue (10 bytes)  inline (hot)
```

Conceptos

Método hash tras hacer inlining

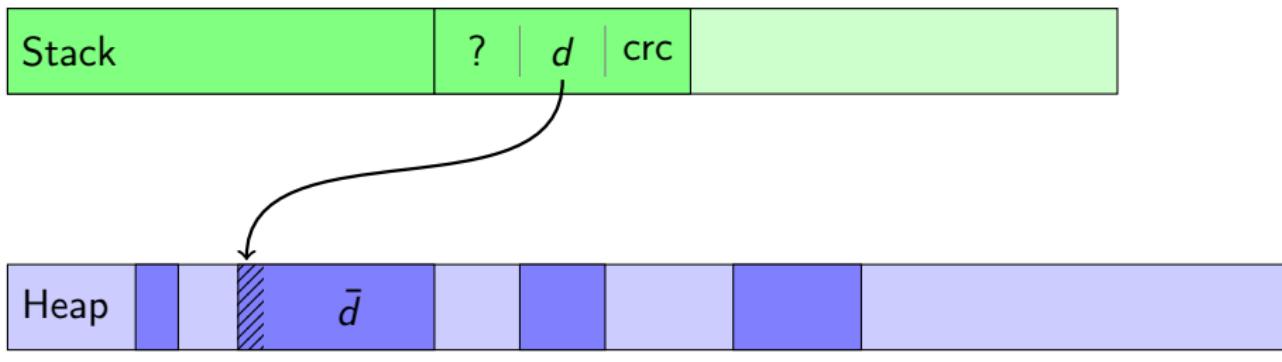
```
1 long hash(byte[] d) {  
2     var h = new Object() {int crc;};  
3     h.crc = 0;  
4     h.crc = updateBytes0(h.crc, d, 0, d.length);  
5     return (long)h.crc & 0xffffffffL;  
6 }
```



Conceptos

Método hash tras hacer Scalar Replacement

```
1 long hash(byte[] d) {  
2     int crc = 0;  
3     crc = updateBytes0(crc, d, 0, d.length);  
4     return (long)crc & 0xffffffffL;  
5 }
```



Scalar Replacement

- Es una transformación más profunda
- El ámbito de aplicación es más limitado, porque exige que el valor no escape en ningún **sentido**
- No afecta al GC
- Los campos no tienen por qué estar consecutivos, y podrían incluso alojarse en registros del procesador.
- En definitiva es un *caso especial* de Stack Allocation pero no una *mejora*

Definiciones

- Escape Analysis: Determina el ámbito dinámico de un objeto
- Partial Escape Analysis: Variante más sofisticada que es capaz de operar sobre bifurcaciones en el código
- Stack Allocation: Coloca objetos *completos* en la pila. Sólo se puede hacer si no sobrevive al método
- Scalar Replacement: Coloca los campos de un objeto en la pila.

Conceptos

Estado del arte

- OpenJDK (Hotspot C2) Escape Analysis, Scalar Replacement para objetos
- GraalVM Partial Escape Analysis, Scalar Replacement para objetos, Scalar replacement limitado para arrays
- OpenJ9 Partial Escape Analysis, **Stack Allocation** para objetos, Scalar Replacement para objetos
- Azul Zing Partial Escape Analysis, Scalar Replacement para objetos, Scalar replacement limitado para arrays

Algoritmos de hashing

Definición y actores

- Dado un dato de tamaño arbitrario produce un resumen de N bits
- Las garantías del resumen tienen diversos niveles: dispersión, dificultad de colisión, uso criptográfico
- Multitud de actores con variantes CRC32, Adler, MurmurHash (1, 2 y 3), xxHash (XXH32, XXH64, XXH3_64, XXH3_128), CityHash, MetroHash, SipHash, ...
- Siempre trabajan igual
 - ▶ Descomponen la entrada en bloques de tamaño fijo + una cola
 - ▶ Procesan los bloques secuencialmente
 - ▶ Hay un proceso de cierre, en el que suele intervenir la longitud total, y que produce el resumen

Algoritmos de hashing

Planteamientos usuales: hashing para diversos tipos

```
1  public interface Hashing {  
2      long hash(byte x);  
3      long hash(char x);  
4      long hash(short x);  
5      long hash(int x);  
6      long hash(long x);  
7      long hash(float x);  
8      long hash(double x);  
9  
10     long hash(byte[] xs);  
11     long hash(byte[] xs, int off, int len);  
12  
13     long hash(char[] xs);  
14     long hash(char[] xs, int off, int len);  
15     long hash(String xs);  
16     long hash(String xs, int off, int len);  
17 }
```

Algoritmos de hashing

Planteamientos usuales: hashing incremental

```
1  public interface Hasher {  
2      Hasher add(byte x);  
3      Hasher add(char x);  
4      Hasher add(short x);  
5      Hasher add(int x);  
6      Hasher add(long x);  
7      Hasher add(float x);  
8      Hasher add(double x);  
9      Hasher add(byte[] xs);  
10     Hasher add(byte[] xs, int off, int len);  
11     Hasher add(char[] xs);  
12     Hasher add(char[] xs, int off, int len);  
13     Hasher add(String xs);  
14     Hasher add(String xs, int off, int len);  
15  
16     long hash64();  
17     int hash(long[] h);  
18 }
```

Algoritmos de hashing

Los Hasher se implementan usualmente con un buffer

```
1  public class AbstractHasher implements Hasher {  
2      private byte[] buf;  
3      private int used;  
4  
5      protected AbstractHasher() {  
6          this.buf = new byte[BLOCK_SIZE+7];  
7          this.used = 0;  
8      }  
9      ...  
10     public Hasher add(int x) {  
11         Bits.le32(buf, used, x);  
12         used += 4;  
13         mayFlush();  
14         return this;  
15     }  
16     ...  
17 }
```

Algoritmos de hashing

¿Cómo se implementa un Hashing?

- A través de un Hasher, salvo para byte[] (Guava)
- Especializando el código del algoritmo a mano para cada tipo (Zero-Allocation-Hashing)

Algoritmos de hashing

Hashish: el experimento fumeta

- Hashish es una librería de hashing que intenta congregar (1) un diseño con descomposición de responsabilidades con (2) el mejor rendimiento posible en Java
- Para ello se basa fuertemente en Scalar Replacement
- Introduce el concepto de Kernel

Algoritmos de hashing

Hashish: Un kernel para cada tamaño de bloque

```
1 public interface Kernel64 {  
2     void block(long block);  
3     void tail(long tail, int taillen, long total);  
4 }  
5  
6 public interface Kernel128 {  
7     void block(long b0, long b1);  
8     void tail(long b0, long b1, int taillen, long total);  
9 }
```

Hashish: Los kernels se crean a cascoporro

```
1  public abstract class HashingKernel64 implements Hashing {  
2      protected abstract Kernel64 newKernel();  
3      public long hash(byte x) {  
4          Kernel64 kernel = newKernel();  
5          kernel.tail(x, 1, 1);  
6          return kernel.hash();  
7      }  
8      ...  
9      public long hash(int x) {  
10         Kernel64 kernel = newKernel();  
11         kernel.tail(x, 4, 4);  
12         return kernel.hash();  
13     }  
14     public long hash(long x) {  
15         Kernel64 kernel = newKernel();  
16         kernel.block(x);  
17         kernel.tail(0, 0, 8);  
18         return kernel.hash();  
19     }  
}
```

Algoritmos de hashing

Hashish: Los kernels se crean a cascoporro 2

```
1      ...
2
3  public long hash(byte[] xs, int off, int len) {
4      final Kernel64 kernel = newKernel();
5      final int blockno = len / 8;
6      for (int i = 0; i < blockno; i++) {
7          kernel.block(Bits.le64(xs, off + 8*i));
8      }
9      final int taillen = len - 8*blockno;
10     kernel.tail(Bits.le64tail(xs, off + 8*blockno, taillen),
11                   taillen, len);
12     return kernel.hash();
13 }
```

Algoritmos de hashing

Hashish: Para un nuevo algoritmo basta un nuevo Kernel

```
1  public class SipHashKernel implements Kernel64 {  
2      private long v0, v1, v2, v3;  
3  
4      public void block(long block) {  
5          v3 ^= block;  
6          compressionRounds();  
7          v0 ^= block;  
8      }  
9  
10     public void tail(long tail, int taillen, long totallen) {  
11         final long mask = ~((~0L) << (8*taillen));  
12         block((mask & tail) | ((totallen & 0xFF) << 56));  
13         v2 ^= 0xFF;  
14         finalizationRounds();  
15     }  
16     ...  
17 }
```

Algoritmos de hashing

Hashish: ... y definir el Hashing

```
1  public class SipHashing extends HashingKernel64 {  
2      private final long k0, k1;  
3  
4      public SipHashing(long k0, long k1) {  
5          this.k0 = k0;  
6          this.k1 = k1;  
7      }  
8  
9      protected SipHashKernel newKernel() {  
10         return new SipHashKernel(k0,k1);  
11     }  
12 }
```

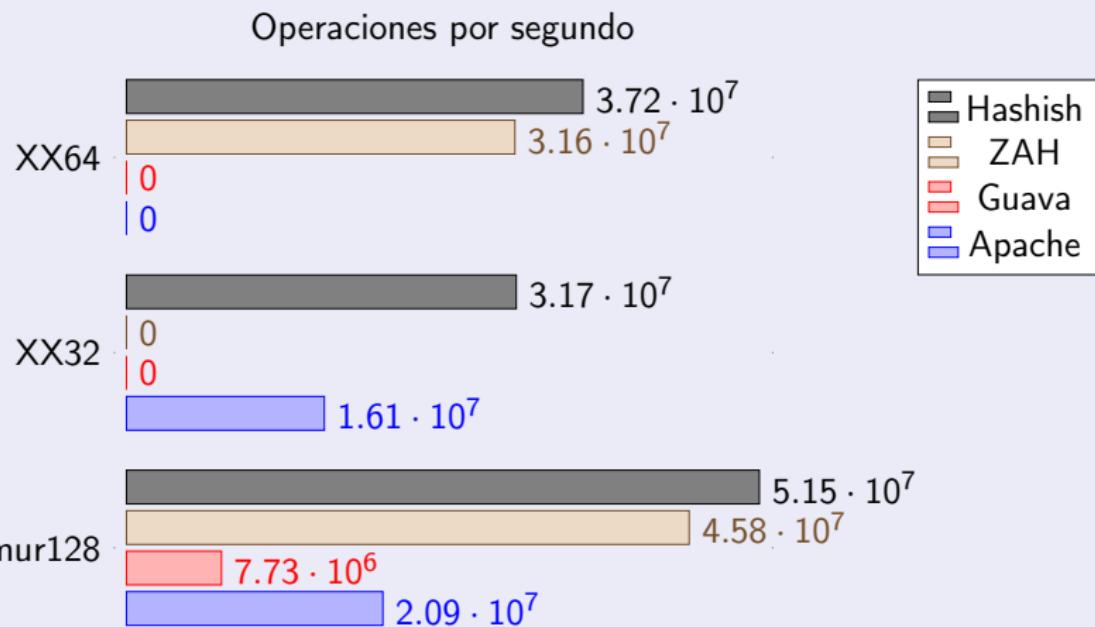
Algoritmos de hashing

Rendimiento sobre byte[3]



Algoritmos de hashing

Rendimiento sobre byte[31]



Algoritmos de hashing

Rendimiento sobre byte[1048607]



Testing de Scalar Replacement

Dudas

- ¿Cómo comprobar si la JVM descubre lo que *claramente nos parece* un candidato a scalar replacement?
- ¿Deberíamos testear scalar replacement?
- ¿Deberíamos testear el rendimiento?
- ¿Y ser capaces de detectar regresiones de rendimiento?
- Scalar replacement es una optimización tachada de *esquiva* que además depende de inlining que es *incierta*
- JMH + inspección visual, de OPS o de GC

Testing de Scalar Replacement

Objetivo

Poder escribir un test (unitario) que compruebe que una sección de código no crea objetos de un juego de clases

Testing de Scalar Replacement

Objetivo

Poder escribir un test (unitario) que compruebe que una sección de código no crea objetos de un juego de clases (**posiblemente tras unas cuantas iteraciones**)

Testing de Scalar Replacement

Possible idea: estadísticas de memoria usada

- `MemoryMXBean.getHeapMemoryUsage()` da la cantidad de memoria usada en el heap
- `GarbageCollectorMXBean.getCollectionCount()` da la cantidad de GCs que ha habido
- Combinando ambas podemos ejecutar una sección de código y comprobar que (1) no hubo GC y (2) cómo aumentó la memoria
- Inconveniente: No permite distinguir qué objetos se han creado. Sólo es útil si nuestro objeto es el dominante
- Descarte: los datos de consumo de memoria no se actualizan inmediatamente

Testing de Scalar Replacement

Possible idea: listener de GC

- `GarbageCollectorMXBean.addNotificationListener()` permite escuchar el resultado de cada GC
- `MemoryMXBean.gc()` fuerza un GC
- Combinando ambas se puede hacer una limpieza (forzar un GC y escuchar la información de su fin), ejecutar nuestra sección de código, volver a hacer una limpieza. La basura de la segunda limpieza la habrá creado nuestra sección de código
- Inconveniente: No permite distinguir qué objetos se han creado. Sólo es útil si nuestro objeto es el dominante
- Descarte: la propia medida genera basura, en cantidad que depende mucho del algoritmo de GC

Testing de Scalar Replacement

Possible idea: Monstruo 2: hprof

- `HotSpotDiagnosticMXBean.dumpHeap` permite hacer un volcado de la memoria en formato hprof
- `HeapFactory.createHeap` de NetBeans permite analizar un hprof
- Combinadas dan lugar al siguiente esquema
 - ① Forzamos GC
 - ② Hacemos volcado inicial, contamos el número I_c de instancias de las clases que nos interesan y descartamos el volcado
 - ③ Ejecutamos nuestra sección de código
 - ④ Hacemos volcado final
 - ⑤ Comprobamos que no ha habido ningún GC
 - ⑥ Contamos en el volcado final el número F_c de instancias de las clases que nos interesan
 - ⑦ $F_c - I_c$ son las instancias creadas

CRC32: ejemplo de test

```
1  @Test void crc32EventuallyDoesNotAllocate() {
2      final byte[] data = new byte[2*4];
3      var chk = new TryAllocationChecker(CRC32.class).verbose();
4      for (int i = 0; i < 25 && !chk.satisfied(); i++) {
5          try (var ignored = chk.enter()) {
6              for (int j = 0; j < 10_000; j++) {
7                  Bits.le32(data, 0, i); Bits.le32(data, 4, j);
8                  chk.consume(
9                      crc32(data));
10             }
11         }
12     }
13     chk.assertSatisfied();
14 }
15 long crc32(byte[] data) {
16     var her = new CRC32();
17     her.update(data);
18     return her.getValue();
19 }
```

Testing de Scalar Replacement

CRC32: ejecución test

```
1 AllocationCheckerTest > crc32EventuallyDoesNotAllocate()
2     Allocation: [gcs: 0, instances: CRC32=10066]
3     Allocation: [gcs: 0, instances: CRC32=0]
4     Allocation: [gcs: 0, instances: CRC32=0]
5     Allocation: [gcs: 0, instances: CRC32=0]
6     Reduced: 85899345900000
```

Testing de Scalar Replacement

String concat: ejemplo de test

```
1  @Test void concatEventuallyDoesNotAllocateStringBuilder() {  
2      var chk = new TryAllocationChecker(StringBuilder.class);  
3      for (int i = 0; i < 25 && !chk.satisfied(); i++) {  
4          try (var ignored = chk.enter()) {  
5              for (int j = 0; j < 10_000; j++) {  
6                  chr.consume(message(i,j));  
7              }  
8          }  
9      }  
10     chk.assertSatisfied();  
11 }  
12  
13 int message(int i, int j) {  
14     return ("Esos " + i + " tipos con bigote tienen cara de "  
15         + j + " otentotes").length();  
16 }
```

Testing de Scalar Replacement

String concat: ejecución test

```
1 AllocationCheckerTest >
    concatEventuallyDoesNotAllocateStringBuilder()
2     Allocation: [gcs: 0, instances: StringBuilder=682]
3     Allocation: [gcs: 0, instances: StringBuilder=13]
4     Allocation: [gcs: 0, instances: StringBuilder=13]
5     Allocation: [gcs: 0, instances: StringBuilder=12]
6 Reduced: 2155560
```

Testing de Scalar Replacement

StringBuilder: ejemplo de test

```
1 int message(int i, int j) {  
2     return new StringBuilder()  
3         .append("Esos ")  
4         .append(i)  
5         .append(" tipos con bigote tienen cara de ")  
6         .append(j)  
7         .append(" otentotes")  
8         .toString()  
9         .length();  
10 }
```

Testing de Scalar Replacement

String concat: ejecución test

```
1 AllocationCheckerTest >
    concatEventuallyDoesNotAllocateStringBuilder()
2     Allocation: [gcs: 0, instances: StringBuilder=10634]
3     Allocation: [gcs: 0, instances: StringBuilder=13]
4     Allocation: [gcs: 0, instances: StringBuilder=12]
5     Allocation: [gcs: 0, instances: StringBuilder=12]
6     Reduced: 2155560
```

Testing de Scalar Replacement

Cálculo iterativo: ejemplo de limitaciones

```
1 long power(long n, long m) {  
2     Long p = new Long(0);  
3     Long b = new Long(m);  
4     while (b.longValue() > 0) {  
5         p = new Long(p.longValue() * n);  
6         b = new Long(b.longValue() - 1);  
7     }  
8     return p.longValue();  
9 }
```

Testing de Scalar Replacement

Cálculo iterativo: ejecución test

```
1 AllocationCheckerTest > powerEventuallyDoesNotAllocate()
2     Allocation: [gcs: 0, instances: Long=193294]
3     Allocation: [gcs: 0, instances: Long=194533]
4     ...
5     Allocation: [gcs: 0, instances: Long=194434]
6
7 AllocationCheckerTest > powerEventuallyDoesNotAllocate()
    FAILED
8     org.opentest4j.AssertionFailedError: expected: <true> but
        was: <false>
```

Testing de Scalar Replacement

$3n + 1$: ejemplo de limitaciones

```
1 long collatzConjecture(long n) {  
2     Long r;  
3     if (n % 2 == 0) {  
4         r = new Long(n / 2);  
5     } else {  
6         r = new Long(3*n + 1);  
7     }  
8     return r.longValue();  
9 }
```

Testing de Scalar Replacement

String concat: ejecución test

```
1 AllocationCheckerTest >
    collatzConjectureEventuallyDoesNotAllocate()
2     Allocation: [gcs: 0, instances: Long=143168]
3     Allocation: [gcs: 0, instances: Long=107360]
4     ...
5     Allocation: [gcs: 0, instances: Long=107293]
6
7 AllocationCheckerTest >
    collatzConjectureEventuallyDoesNotAllocate() FAILED
8     org.opentest4j.AssertionFailedError: expected: <true>
        but was: <false>
```

Efecto del polimorfismo

Monstruo 3: el polimorfismo

- Un *call site* es un punto de llamada a un método
- Se clasifican en monomórficos, polimórficos y megamórficos en función de la variedad de clases que son destino de esa llamada
- El inlining está limitado por el polimorfismo
- Y el scalar replacement está limitado por el inlining
- Y ... los métodos heredados no cambian de clase

Efecto del polimorfismo

Clase abstracta

```
1  public abstract class HashingKernel64 implements Hashing {  
2      protected abstract Kernel64 newKernel();  
3  
4      ...  
5      public long hash(long x) {  
6          final Kernel64 kernel = newKernel();  
7          kernel.block(x);  
8          kernel.tail(0, 0, 8);  
9          return kernel.hash();  
10         ...  
11     }
```

Efecto del polimorfismo

Uso monomórfico (2)

```
1 static final Hashing H1 = new MyHashing1();
2 static final Hashing H2 = new MyHashing2();
3
4 long ultrahash2(long d) {
5     return
6         H1.hash(d)
7         + H2.hash(d);
8 }
```

Efecto del polimorfismo

Clase abstracta: inlining (2)

```
1  HashingKernel64::hash (30 bytes)  made not entrant
2  @ 1  MyHashing2::newKernel (8 bytes)  inline (hot)
3  @ 1  MyHashing1::newKernel (8 bytes)  inline (hot)
4  \-> TypeProfile (8702/17405 counts) = MyHashing1
5  \-> TypeProfile (8703/17405 counts) = MyHashing2
6    @ 4  MyKernel1::<init> (6 bytes)  inline (hot)
7      @ 2  MyKernel1::<init> (15 bytes)  inline (hot)
8        @ 1  java.lang.Object::<init> (1 bytes)  inline (hot)
9    @ 4  MyKernel2::<init> (6 bytes)  inline (hot)
10   @ 2  MyKernel2::<init> (15 bytes)  inline (hot)
11     @ 1  java.lang.Object::<init> (1 bytes)  inline (hot)
12   @ 7  MyKernel2::block (11 bytes)  inline (hot)
13   @ 7  MyKernel1::block (11 bytes)  inline (hot)
14   \-> TypeProfile (8702/17405 counts) = MyKernel1
15   \-> TypeProfile (8703/17405 counts) = MyKernel2
16   @ 18  MyKernel2::tail (14 bytes)  inline (hot)
17   @ 18  MyKernel1::tail (14 bytes)  inline (hot)
18   \-> TypeProfile (8702/17405 counts) = MyKernel1
19   \-> TypeProfile (8703/17405 counts) = MyKernel2
20   @ 24  MyKernel2::hash (5 bytes)  accessor
21   @ 24  MyKernel1::hash (5 bytes)  accessor
22   \-> TypeProfile (8702/17405 counts) = MyKernel1
23   \-> TypeProfile (8703/17405 counts) = MyKernel2
```

Uso monomórfico: inlining (2)

```
1 ultrahash2 @ 54 (147 bytes)    made not entrant
2 @ 75 HashingKernel64::hash (30 bytes)    inline (hot)
3 @ 1 MyHashing1::newKernel (8 bytes)    inline (hot)
4 ...
5 @ 7 MyKernel2::block (11 bytes)    inline (hot)
6 @ 7 MyKernel1::block (11 bytes)    inline (hot)
7 \-> TypeProfile (13620/27241 counts) = MyKernel1
8 \-> TypeProfile (13621/27241 counts) = MyKernel2
9 @ 18 MyKernel2::tail (14 bytes)    inline (hot)
10 @ 18 MyKernel1::tail (14 bytes)    inline (hot)
11 \-> TypeProfile (13620/27241 counts) = MyKernel1
12 \-> TypeProfile (13621/27241 counts) = MyKernel2
13 @ 24 MyKernel2::hash64 (5 bytes)    accessor
14 @ 24 MyKernel1::hash64 (5 bytes)    accessor
15 \-> TypeProfile (13620/27241 counts) = MyKernel1
16 \-> TypeProfile (13621/27241 counts) = MyKernel2
17 @ 85 HashingKernel64::hash (30 bytes)    inline (hot)
18 @ 1 MyHashing2::newKernel (8 bytes)    inline (hot)
19 ...
20 @ 7 MyKernel2::block (11 bytes)    inline (hot)
21 @ 7 MyKernel1::block (11 bytes)    inline (hot)
22 \-> TypeProfile (13620/27241 counts) = MyKernel1
23 \-> TypeProfile (13621/27241 counts) = MyKernel2
24 @ 18 MyKernel2::tail (14 bytes)    inline (hot)
25 @ 18 MyKernel1::tail (14 bytes)    inline (hot)
26 \-> TypeProfile (13620/27241 counts) = MyKernel1
27 \-> TypeProfile (13621/27241 counts) = MyKernel2
28 @ 24 MyKernel2::hash64 (5 bytes)    accessor
29 @ 24 MyKernel1::hash64 (5 bytes)    accessor
30 \-> TypeProfile (13620/27241 counts) = MyKernel1
31 \-> TypeProfile (13621/27241 counts) = MyKernel2
```

Efecto del polimorfismo

Uso monomórfico (3)

```
1 static final Hashing H1 = new MyHashing1();
2 static final Hashing H2 = new MyHashing2();
3 static final Hashing H3 = new MyHashing3();
4
5 long ultrahash3(long d) {
6     return
7         H1.hash(d)
8         + H2.hash(d)
9         + H3.hash(d);
10 }
```

Efecto del polimorfismo

Clase abstracta: inlining (3)

```
1 HashingKernel64::hash (30 bytes)  made not entrant
2 @ 1  HashingKernel64::newKernel (0 bytes)  failed to inline: virtual call
3 @ 7  Kernel64::block (0 bytes)  failed to inline: virtual call
4 @ 18  Kernel64::tail (0 bytes)  failed to inline: virtual call
5 @ 24  Hash::hash64 (10 bytes)  failed to inline: virtual call
```

Efecto del polimorfismo

Uso monomórfico: inlining (3)

```
1 ultrahash3 @ 54 (158 bytes)    made not entrant
2 @ 75   HashingKernel64::hash (30 bytes)    inline (hot)
3   @ 1   MyHashing1::newKernel (8 bytes)    inline (hot)
4     @ 4   MyKernel1::<init> (6 bytes)    inline (hot)
5       @ 2   MyKernel1::<init> (15 bytes)   inline (hot)
6         @ 1   java.lang.Object::<init> (1 bytes)   inline (hot)
7   @ 7   Kernel64::block (0 bytes)    virtual call
8   @ 18  Kernel64::tail (0 bytes)    virtual call
9   @ 24  Hash::hash64 (10 bytes)    virtual call
10  @ 85  HashingKernel64::hash (30 bytes)   inline (hot)
11  @ 1   MyHashing2::newKernel (8 bytes)   inline (hot)
12    @ 4   MyKernel2::<init> (6 bytes)   inline (hot)
13      @ 2   MyKernel2::<init> (15 bytes)  inline (hot)
14        @ 1   java.lang.Object::<init> (1 bytes)  inline (hot)
15   @ 7   Kernel64::block (0 bytes)    virtual call
16   @ 18  Kernel64::tail (0 bytes)    virtual call
17   @ 24  Hash::hash64 (10 bytes)    virtual call
18  @ 96  HashingKernel64::hash (30 bytes)   inline (hot)
19  @ 1   MyHashing3::newKernel (8 bytes)   inline (hot)
20    @ 4   MyKernel3::<init> (6 bytes)   inline (hot)
21      @ 2   MyKernel3::<init> (15 bytes)  inline (hot)
22        @ 1   java.lang.Object::<init> (1 bytes)  inline (hot)
23   @ 7   Kernel64::block (0 bytes)    virtual call
24   @ 18  Kernel64::tail (0 bytes)    virtual call
25   @ 24  Hash::hash64 (10 bytes)    virtual call
```

Efecto del polimorfismo

Monstruo 4: el splitting

- Splitting: Intenta eliminar los puntos polimórficos buscando un punto de llamada anterior monomórfico
- ¿Lo tiene alguna JVM?
- Pero un inlining *más precavido* tiene un efecto similar

Efecto del polimorfismo

Uso monomórfico: C2 v21 al rescate (2)

```
1  ultrahash2 @ 54 (147 bytes)    made not entrant
2  @ 75   HashingKernel64::hash (30 bytes)  inline (hot)
3  @ 1    MyHashing2::newKernel (8 bytes)  inline (hot)
4  @ 4    MyKernel1::<init> (6 bytes)  inline (hot)
5  @ 2    MyKernel1::<init> (15 bytes) inline (hot)
6  @ 1    java.lang.Object::<init> (1 bytes)  inline (hot)
7  @ 7    MyKernel1::block (11 bytes)  inline (hot)
8  @ 18   MyKernel1::tail (14 bytes)  inline (hot)
9  @ 24   MyKernel1::hash64 (5 bytes)  accessor
10 @ 85  HashingKernel64::hash (30 bytes)  inline (hot)
11 @ 1    MyHashing2::newKernel (8 bytes)  inline (hot)
12 @ 4    MyKernel2::<init> (6 bytes)  inline (hot)
13 @ 2    MyKernel2::<init> (15 bytes) inline (hot)
14 @ 1    java.lang.Object::<init> (1 bytes)  inline (hot)
15 @ 7    MyKernel2::block (11 bytes)  inline (hot)
16 @ 18   MyKernel2::tail (14 bytes)  inline (hot)
17 @ 24   MyKernel2::hash64 (5 bytes)  accessor
```

Efecto del polimorfismo

Uso monomórfico: C2 v21 al rescate (3)

```
1  ultrahash3 @ 54 (158 bytes)  made not entrant
2  @ 75  HashingKernel64::hash (30 bytes)  inline (hot)
3  @ 1  MyHashing3::newKernel (8 bytes)  inline (hot)
4  @ 4  MyKernel1::<init> (6 bytes)  inline (hot)
5  @ 2  MyKernel1::<init> (15 bytes)  inline (hot)
6  @ 1  java.lang.Object::<init> (1 bytes)  inline (hot)
7  @ 7  MyKernel1::block (11 bytes)  inline (hot)
8  @ 18  MyKernel1::tail (14 bytes)  inline (hot)
9  @ 24  MyKernel1::hash64 (5 bytes)  accessor
10 @ 85  HashingKernel64::hash (30 bytes)  inline (hot)
11 @ 1  MyHashing3::newKernel (8 bytes)  inline (hot)
12 @ 4  MyKernel2::<init> (6 bytes)  inline (hot)
13 @ 2  MyKernel2::<init> (15 bytes)  inline (hot)
14 @ 1  java.lang.Object::<init> (1 bytes)  inline (hot)
15 @ 7  MyKernel2::block (11 bytes)  inline (hot)
16 @ 18  MyKernel2::tail (14 bytes)  inline (hot)
17 @ 24  MyKernel2::hash64 (5 bytes)  accessor
18 @ 96  HashingKernel64::hash (30 bytes)  inline (hot)
19 @ 1  MyHashing3::newKernel (8 bytes)  inline (hot)
20 @ 4  MyKernel3::<init> (6 bytes)  inline (hot)
21 @ 2  MyKernel3::<init> (15 bytes)  inline (hot)
22 @ 1  java.lang.Object::<init> (1 bytes)  inline (hot)
23 @ 7  MyKernel3::block (11 bytes)  inline (hot)
24 @ 18  MyKernel3::tail (14 bytes)  inline (hot)
25 @ 24  MyKernel3::hash64 (5 bytes)  accessor
```

Polimorfismo intrínseco

Recodificación de datos

```
1 interface Parser {  
2     Object parse(String val);  
3 }  
4  
5 interface Marshaler {  
6     void marshal(ByteBuffer bb, Object v);  
7 }
```

Polimorfismo intrínseco

Recodificación de datos

```
1 class IntParser implements Parser {  
2     Object parse(String val) {  
3         return new Integer(Integer.parseInt(val));  
4     }  
5 }  
6  
7 class IntMarshaler implements Marshaler {  
8     void marshal(ByteBuffer bb, Object v) {  
9         bb.putInt((int) v);  
10    }  
11 }
```

Polimorfismo intrínseco

Recodificación de datos dirigida por tipos

```
1 void transfer(String[] types,
2                 String[] vals, ByteBuffer trg) {
3     for (int i = 0; i < vals.length; i++) {
4         var par = UNIV.parserFor(types[i]);
5         var mar = UNIV.marshalerFor(types[i]);
6         mar.marshal(trg, par.parse(vals[i]))
7     }
8 }
```

Polimorfismo intrínseco

Aplicación parcial de la recodificación

```
1 interface WidePipe {  
2     transfer(String[] vals, ByteBuffer trg);  
3 }  
4  
5 WidePipe transfer(String[] types) {  
6     var pars = UNIV.parsersFor(types);  
7     var mars = UNIV.marshalersFor(types);  
8     return (vals, trg) -> {  
9         for (int i = 0; i < vals.length; i++) {  
10             mars[i].marshal(trg, pars[i].parse(vals[i]));  
11         }  
12     };  
13 }
```

Generalización de la aplicación parcial

```
1 interface Pipe {
2     transfer(String val, ByteBuffer trg);
3 }
4 Pipe pipe(Parser par, Marshaler mar) {
5     return (val, trg) -> mar.marshal(trg, par.parse(val));
6 }
7 Pipe pipeFor(String type) {
8     return pipe(UNIV.parserFor(type), UNIV.marshalerFor(type));
9 }
10 Pipe[] pipesFor(String[] types) {
11     var pipes = new Pipe[types.length];
12     for (int i = 0; i < types.length; i++) {
13         pipes[i] = pipeFor(types[i]);
14     }
15     return pipes;
16 }
17 WidePipe compose(Pipe[] pipes) {
18     return (vals, trg) -> {
19         for (int i = 0; i < vals.length; i++) {
20             pipes[i].transfer(vals[i], trg)
21         }
22     };
23 }
24 WidePipe transfer(String[] types) {
25     return compose(pipesFor(types));
26 }
```

Polimorfismo intrínseco

Fuente del polimorfismo intrínseco

```
1 Pipe pipe(Parser par, Marshaler mar) {  
2     return (val, trg) -> mar.marshal(trg, par.parse(val));  
3 }
```

Polimorfismo intrínseco

Fuente del polimorfismo intrínseco (modo clásico)

```
1 Pipe pipe(Parser par, Marshaler mar) {  
2     return new PMPipe(par, mar);  
3 }  
4  
5 class PMPipe implements Pipe {  
6     Parser par;  
7     Marshaler mar;  
8     PMPipe(Parser par, Marshaler mar) {  
9         this.par = par;  
10        this.mar = mar;  
11    }  
12    void transfer(String val, ByteBuffer trg) {  
13        mar.marshal(trg, par.parse(val));  
14    }  
15 }
```

Monstruo 5:

- Hay que generar una implementación de **Pipe** para cada tipo
- Siempre sería el código exacto de **PMPipe**
- Soluciones

Monstruo 5: ClassLoader

- Hay que generar una implementación de `Pipe` para cada tipo
- Siempre sería el código exacto de `PMPipe`
- Soluciones
 - ▶ A mano
 - ▶ Generación de bytecode: ASM, class-file API, ...
 - ▶ `java.lang.invoke.LambdaMetafactory`
 - ▶ `ClassLoader`

Polimorfismo intrínseco

Pipe factory

```
1 class PipeFactory {  
2     Map<String,Constructor<Pipe>> byType = new HashMap<>();  
3     BoundedSpecializer spec  
4         = new BoundedSpecializer(Pipe.class);  
5  
6     Pipe pipeFor(String type, Parser par, Marshaler mar) {  
7         Constructor<Pipe> con = byType.computeIfAbsent(type,  
8             key -> spec.specialized(PMPipe.class)  
9                 .getConstructor(Parser.class, Marshaler.class));  
10        return (Pipe) con.newInstance(par, mar);  
11    }  
12 }
```

Polimorfismo intrínseco

Transferencia del polimorfismo

```
1 Pipe pipeFor(String type) {  
2     return PipeFactory.getInstance().pipeFor(  
3         type, UNIV.parserFor(type), UNIV.marshalerFor(type));  
4 }  
5  
6 WidePipe compose(Pipe[] pipes) {  
7     return (vals, trg) -> {  
8         for (int i = 0; i < vals.length; i++) {  
9             pipes[i].transfer(vals[i], trg)  
10        }  
11    };  
12 }
```

Polimorfismo intrínseco

BoundedSpecializer

```
1  class BoundedSpecializer {  
2      ClassSet toSpecialize;  
3      BoundedSpecializer(Class<?>... root) {  
4          this.toSpecialize = new HierarchyClassSet(root);  
5      }  
6  
7      Class<?> specialized(Class<?> klass) {  
8          return toSpecialize.contains(klass)  
9              ? reload(klass.getName())  
10             : unmanagedClassError(klass.getName());  
11      }  
12  
13      Class<?> reload(String classname)  
14      throws ClassNotFoundException {  
15          return new SpecializingClassLoader(toSpecialize)  
16              .loadClass(classname);  
17      }  
18  }
```

SpecializingClassLoader

```
1 class SpecializingClassLoader extends ClassLoader {  
2     ClassSet toLoad;  
3     SpecializingClassLoader(ClassSet toLoad) {  
4         this.toLoad = toLoad;  
5     }  
6  
7     Class<?> loadClass(String name, boolean resolve) {  
8         Class<?> klass = findLoadedClass(name);  
9         if (klass == null) {  
10             klass = toLoad.contains(name) ? loadCopy(name)  
11                 : super.loadClass(name, false);  
12         }  
13         if (resolve) resolveClass(klass);  
14         return klass;  
15     }  
16  
17     Class<?> loadCopy(String name) {  
18         byte[] bytecode = readResource(name);  
19         return defineClass(name, bytecode, 0, bytecode.length);  
20     }  
21 }
```

Polimorfismo intrínseco

Parcheando C2 v17

```
1 static final Hashing H1;
2 static final Hashing H2;
3 static final Hashing H3;
4 static {
5     var spec = new BoundedSpecializer(Hashing.class);
6     H1 = (Hashing) spec.specialized(MyHashing1.class)
7         .newInstance();
8     H2 = (Hashing) spec.specialized(MyHashing2.class)
9         .newInstance();
10    H2 = (Hashing) spec.specialized(MyHashing3.class)
11        .newInstance();
12 }
13
14 long ultrahash3(long d) {
15     return h1.hash(d) + h2.hash(d) + h3.hash(d);
16 }
```

Polimorfismo intrínseco

Ejercicio de clase

¿new Integer o Integer.valueOf o nada?

```
1 class IntParser {  
2     Object parse(String val) {  
3         return new Integer(Integer.parseInt(val));  
4     }  
5 }
```

Polimorfismo intrínseco

Ejercicio de clase

¿new Integer o Integer.valueOf o nada?

```
1 class IntParser {  
2     Object parse(String val) {  
3         return Integer.valueOf(Integer.parseInt(val));  
4     }  
5 }
```

Polimorfismo intrínseco

Ejercicio de clase

¿new Integer o Integer.valueOf o nada?

```
1 class IntParser {  
2     Object parse(String val) {  
3         return Integer.parseInt(val);  
4     }  
5 }
```

Polimorfismo intrínseco

Ejercicio de clase

¿new Integer o Integer.valueOf o nada?

```
1 class IntParser {  
2     Object parse(String val) {  
3         return Integer.valueOf(Integer.parseInt(val));  
4     }  
5 }  
  
1 public class Integer {  
2     ...  
3     static Integer valueOf(int i) {  
4         if (i >= IntegerCache.low && i <= IntegerCache.high)  
5             return IntegerCache.cache[i + (-IntegerCache.low)];  
6         return new Integer(i);  
7     }  
8     ...  
9 }
```

Polimorfismo intrínseco

Ejercicio para casa

¿Qué pasa si un parseador puede devolver null?

```
1 class IntParser {  
2     Object parse(String val) {  
3         try {  
4             return new Integer(Integer.parseInt(val));  
5         } catch (NumberFormatException e) {  
6             return null;  
7         }  
8     }  
9 }
```

Polimorfismo intrínseco

Ejercicio para casa

Reescribir `compose` para que la llamada a `transfer` sea monomórfica

```
1 Pipe pipeFor(String type) {  
2     return PipeFactory.getInstance().pipeFor(  
3         type, UNIV.parserFor(type), UNIV.marshalerFor(type));  
4 }  
5  
6 WidePipe compose(Pipe[] pipes) {  
7     return (vals, trg) -> {  
8         for (int i = 0; i < vals.length; i++) {  
9             pipes[i].transfer(vals[i], trg)  
10        }  
11    };  
12 }
```

Proyectos de esta presentación

stack-allocation Notas sobre stack allocation. Incluye esta presentación
<https://github.com/SeismoTech/stack-allocation>

Hashish Algoritmos de hashing
<https://github.com/SeismoTech/hashish>

Testafeo Testing de la creación de objetos en el heap
<https://github.com/SeismoTech/testafeo>

LaEspe La Especialización de clases
<https://github.com/SeismoTech/laespe>

Tertulia

