Efficiently Verifiable Escape Analysis

Matthew Q. Beers, Christian H. Stork,
and Michael Franz
School of Information and Computer Science
University of California, Irvine

Presented by
Irantha Suwandarathna

Agenda

- **Motivation**
- **Optimizations with Escape analysis**
- **Analysis**
  - Class transformation
  - Rtt constraints
  - Esc constraints
- **Evaluation**
  - Captured allocations
  - Verification speed

Motivation

- Mobile ,JIT compilers can’t afford time consuming optimizations
- Annotate class files with the analysis results
- Annotations are unsafe
- So need efficient verification

What they achieved

- Linear time analysis
- Significant analysis precision
- Very low annotation overhead
- Easy to verify
- Support dynamic class loading
  - Only a conservative assumption ??

Optimizations with Escape Analysis

- **Stack allocation**
- **Remove synchronization**
- **Object inlining**
- **Dead store removal**

Analysis

- **Find captured variable instead of objects**
  - Never returned
  - Passed only to captured parameters
  - Never assigned to a escaping variables
- **Assume all field references escape**
- **Array elements always escape**
- **Multi-dimension array is captured in the first dimension**

Analysis Steps

- Source program transformation
- Runtime type constrains
- Escape constraints

Source Code Transformations

Run-Time Type Constraints

- **For each variable calculate rtt(v)**
  - uninitialized ($\perp$),initialized but unknown (T),class C
- **Initialize with rtt(v) >= $\perp$**
- **Linear time Solution with standard**

Escape Constraints

- For each variable define Escape constraint
  - esc(v) = T if true, esc(v) = $\perp$ if false

Escape Constraints ...

- Class hierarchy and rtt(v) to find invocable methods

Annotations / Verification

- Generation of constraint equations at run time
- Verify with annotated solutions
- Can’t notice suboptimal solutions
- Library method parameter should be annotated
  - Revert to everything escape

Evaluation

- **Compared with most precise know analysis**
  - Whaley - and Rinard
- **Section 2 and 3 of JavaGrande and a subset of SPECjvm98**
- **string concatenations**
  - String s = s1 + s2;
  - String s = new StringBuffer(s1).append(s2).toString();

Captured Allocation sites (static allocation)

- Difference in Source & bytecode
- Same methods were analysed

Captured allocation sites within loops(closed world)

- better capture of inside loop
- Due to used benchmarks
- Short lived
- high dynamic captured allocations

Why not Dynamic allocations?

- Lack of infrastructure
- Non-trivial to modify VM allocation strategy
- Annotations to guarantee bounded stack
- 1/3 of allocations need dynamic stack frame

Verification Time

- **Can be integrated with another pass**
- **Each method separately**

Annotation Size

- **As *attribute_info* for methods**
- **esc(v) takes 1 bit**
- **rtt(v) as**
  - 4 byte reference to constant pool
  - 1 bit boolean predicate

Encoding rtt(v) as a boolean predicate

- Run-time type equal to declared type(D) or not
- If C != D then replace the declared type with C
- If rtt(v) = $\perp$ replace v with null