Garbage collection in Java

Amer Diwan

Outline

• Java is a statically and strongly typed language
  – Can do accurate garbage collection
  – In particular, can do copying collection
• But, there are aspects of Java that make GC difficult
  – If the language designer has not thought through all implementation implications then it can create unnecessary challenges for implementation
The Java model (simplified)

Java program

Java to bytecodes

- Bytecodes to machine code
- Bytecode interpreter

Semantics of the languages

- There are rules for
  - what is legal Java code
  - what is legal Java bytecodes
- The bytecode verifier checks bytecodes
  - It basically does a type-inference (amongst other things) to ensure that the rules are met
What are java bytecodes like

- Low level language
  - Presumably easy for interpreters to use
  - Stack based: operations take their arguments from the stack and deposit their results on the stack
  - Local variables. Variables in a java program may be mapped to local variables at the bytecode level. At different points in time, the same local variables may be used for different Java level variables

Some rules for bytecodes

- A bytecode is like a machine-independent assembly language
- At every point in the program (with one exception), one can determine if a local variable holds a pointer or a non-pointer
  - Why is this important?
The “exception” to the rule

```java
try
    s = 10;
    ...
    s = new T;
finally
    ...
end
```

What’s the matter here?

What if...

- The finally block accesses accesses ‘s’?
  - Disallowed. Verification will fail since type of ‘s’ is ambiguous
- GC is triggered in the finally block?
  - Don’t know if ‘s’ is a pointer or an integer!
A more elaborate example

- try
  - try
    - s = 10; ...
    - throw have_int;...
    - s = new T; ...
    - throw have_ptr; ...
    - finally ... end
  - except
    - catch(have_int_t v) = ...s...
    - catch(have_ptr_t u) = ...s...
  - end

What’s the deal here?
Java bytecode verification requires polymorphic type inference

A solution to the pointer-nonpointer conflict

- **Split** ambiguous slots into two slots, one for pointer-type and the other for non-pointer-type
- **Analyze** the program to find out conflicts
  - ref-nonref
  - ref-uninit
- Why are ref-uninit conflicts interesting?
An example of the analysis

\[ s = 10 \]
\[ s = \text{new } T \]
\[ \ldots s \ldots \]
\[ \ldots s \ldots \]
\[ s: \{ \text{ref, nonref} \} \]

Splitting based on the analysis

\[ s_1 = 10 \]
\[ s = \text{new } T \]
\[ \ldots s_1 \ldots \]
\[ \ldots s \ldots \]
\[ s: \{ \text{ref} \} \]
\[ s_1: \{ \text{nonref} \} \]

Splitting requires creating a new variable and rewriting code that needs to use the new variable.
A solution to the pointer-unitialized conflict

- Add dummy assignments such that the ambiguous variable are always initialized (to NIL)

```
s = new T
...s...
```

Resolving the ref-uninit conflict

```
s = NIL,
s = new T
...s...
```

```
s: {ref, uninit}
s: {ref}
```
Other issues pointed out in the paper

- liveness analysis
  - If a variable’s value will not be used in the future, then don’t treat it like a pointer even though its current value is a pointer

```
s = new T
...
...s...
...
s = ...
```

Important points

- GC can be hard to do if the implication of language semantics are not fully considered
  - Java was designed to work with GC
  - But it still has some gotchas for GC!
Next topic: Persistent programming languages

• Readings: Atkinson and Morrison (see web page)