

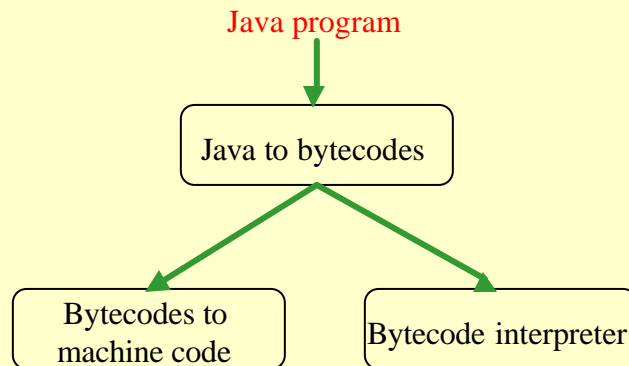
# Garbage collection in Java

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)



## 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

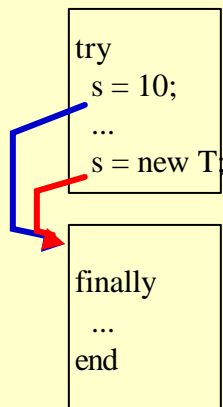
## What are java bytecodes like

- 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



What's the matter here?

## What if...

- The finally block 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

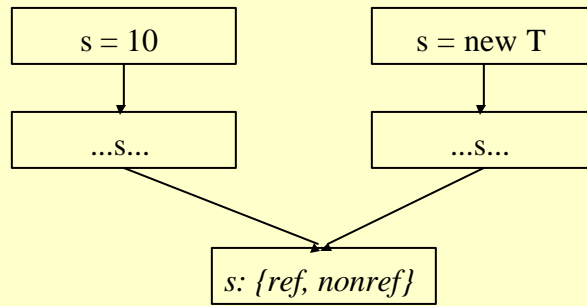
What's the deal here?

Java bytecode verification requires polymorphic type inference

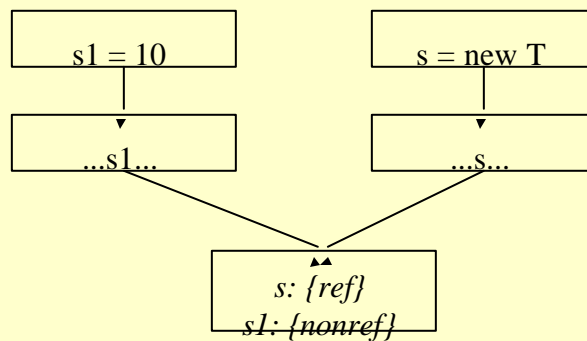
## A solution to the pointer-nonpointer conflict

- 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



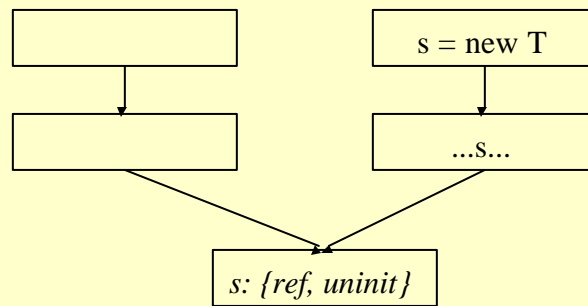
## Splitting based on the analysis



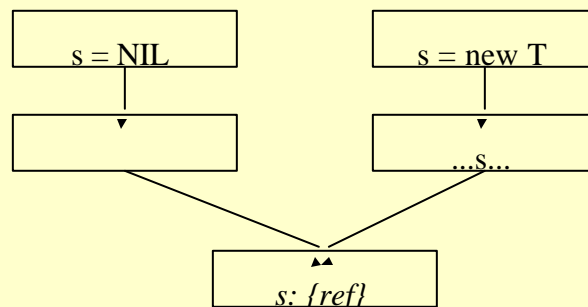
Splitting requires creating a new variable and rewriting code that needs to use the new variable.

## A solution to the pointer-initialized conflict

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

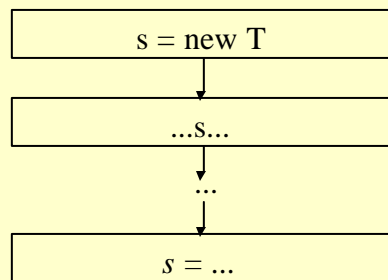


## Resolving the ref-uninit conflict



## 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



## 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)