semantics of Java, revisited

Daniel Jackson

september 17, 2007
a gentle intro with lists
what is espresso?

just a subset of Java

• language is too big to learn in one go
• and too big to define precisely
• standard practice: define a subset

why you should care

• you’ll want to learn other languages
• so what is the essence of Java?
• what are the fundamental ideas?
vars, classes and fields

C v;
class C1 {C2 f;}
class C2 extends C1 {}
class C3 extends C2 {C4 g;}

inheritance

- object inherits fields from its superclasses class
- so object of class C2 or C2 has field f
- but object of class C2 has no field g
example: lists

declarations

List l; Object o1; Object o2;
class List {List reverse;}
class EmptyList extends List {}
class NonEmptyList extends List {Object elt; List rest;}

sample state
espresso: statements

statements

<table>
<thead>
<tr>
<th>statement</th>
<th>syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>constructor</td>
<td>u = new C ();</td>
</tr>
<tr>
<td>assignment</td>
<td>u = v;</td>
</tr>
<tr>
<td>field get</td>
<td>u = v.f;</td>
</tr>
<tr>
<td>field set</td>
<td>u.f = v;</td>
</tr>
</tbody>
</table>

exercise

' write code to create the list of the previous slide
naive type checking

consider this algorithm

\[\begin{align*}
\& \text{associate with each var its declared type} \\
\& \text{allow } x.f \text{ when field } f \text{ belongs to type of } x \text{ or subtype}
\end{align*}\]

why doesn’t this work?

\[\begin{align*}
\& \text{consider} \\
\& \text{List } l; \text{ Object } o; \\
\& o = \text{new Object } (); \\
\& l = \text{new Object } (); \\
\& l.\text{rest} = o;
\end{align*}\]

\[\begin{align*}
\& \text{assignment to } l \text{ “changes” its type!} \\
\& \text{can’t allow such assignments} \\
\& \text{so type checker must impose rules on statements prior to field access}
\end{align*}\]
espresso: type checking rules

logical type assertions

C v;        // in logic, write “v: C”
class C1 {C2 f;}    // in logic, write “f: C1→C2”
class C2 extends C1 {}  // in logic, write “C2 ≤ C1” (and “C2 ≤ C2” etc)
class C3 extends C2 {}  // in logic, write “C3 ≤ C2” (and “C3 ≤ C1” etc)

rules for statements

<table>
<thead>
<tr>
<th>statement</th>
<th>syntax</th>
<th>type checking rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>constructor</td>
<td>u = new C ();</td>
<td>u: cu ⇒ C ≤ cu</td>
</tr>
<tr>
<td>assignment</td>
<td>u = v;</td>
<td>u: cu ∧ v: cv ⇒ cv ≤ cu</td>
</tr>
<tr>
<td>field get</td>
<td>u = v.f;</td>
<td>u: cu ∧ v: cv ∧ f: c→cf ⇒ cv ≤ c ∧ cf ≤ cu</td>
</tr>
<tr>
<td>field set</td>
<td>u.f = v;</td>
<td>... exercise for students ...</td>
</tr>
</tbody>
</table>
formalizing semantics
how do we express the interpreter as a state machine

- as before: just define vars, initialization, operations
- operations are the statements
- hardest part: defining the variables

key idea

- represent the state with relations or tables
  - stack: maps variables to objects
  - heap: maps objects under fields to objects
  - tag: maps objects to type tags
relational state

<table>
<thead>
<tr>
<th>stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>var</td>
</tr>
<tr>
<td>l</td>
</tr>
<tr>
<td>o1</td>
</tr>
<tr>
<td>o2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>heap</th>
</tr>
</thead>
<tbody>
<tr>
<td>ref</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>tag</th>
</tr>
</thead>
<tbody>
<tr>
<td>ref</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
</tbody>
</table>
operators on relations

four operators

• product: make a table with one row
  \[ l \rightarrow 1 \]

• union: just include rows from two tables
  \[ \text{stack} = (l \rightarrow 1) + (o1 \rightarrow 4) + (o2 \rightarrow 5) \]

• override: like union, but replace rows when first column(s) match
  \[ \text{stack} ++ (l \rightarrow 2) = (l \rightarrow 2) + (o1 \rightarrow 4) + (o2 \rightarrow 5) \]

• image: lookup, using value in first column
  \[ \text{stack}[o1] = 4 \]

apply to relations with more than 2 columns too

heap \[ [1][\text{rest}] = 2 \]
state machine semantics

vars
stack: Var -> (Ref + Null)
heap: Ref -> Field -> (Ref + Null)
tag: Ref -> Class

init stack = ∅ && heap = ∅

op constructor (u: Var, C: Class) // u = new C()
post stack' = stack + (u -> r) and tag' = tag + (r -> C) for some fresh r

op assignment (u, v: Var) // u = v
post stack' = stack ++ (u -> stack[v])

op fieldGet (u, v: Var, f: Field) // u = v.f
pre stack[v] != null
post stack' = stack ++ (u -> heap [stack[v]][f])

op fieldSet (u, v: Var, f: Field) // u.f = v
pre stack[u] != null
post heap' = heap ++ (stack[u] -> f) -> stack [v]
type safety (again)
safety invariant

stack safety

- to ensure no bad field refs, just need to ensure that stack is safe
- if variable $x$ has declared type $c$, its reference is tagged with a subtype of $c$

$$\text{STACK-SAFE} \equiv x: c \Rightarrow (\text{stack}[x] = \text{null}) \lor (\text{tag}[\text{stack}[x]] \leq c)$$

- but not an invariant: heap must be safe too

heap safety

- if field $g$ of ref $r$ holds $s$, then $r$ and $s$ have tags compatible with $g$’s decl

$$\text{HEAP-SAFE} \equiv \text{heap}[r][g] = s \land g: c \rightarrow cg \Rightarrow \text{tag}[r] \leq c \land \text{tag}[s] \leq cg$$

invariant

$$\text{INV} \equiv \text{HEAP-SAFE} \land \text{STACK-SAFE}$$
proving the invariant

initially
• stack and heap are empty, so INV holds trivially

preservation
• if INV holds before op, and if op type-checks, then INV holds after

conclusion
• by induction, INV holds after any sequence of ops
preservation arguments

invariant

\[
\text{STACK-SAFE} \equiv \ x: c \Rightarrow (\text{stack}[x] = \text{null}) \lor (\text{tag}[\text{stack}[x]] \leq c)
\]

\[
\text{HEAP-SAFE} \equiv \ \text{heap}[r][g] = s \land g: c \rightarrow cg \Rightarrow \text{tag}[r] \leq c \land \text{tag}[s] \leq cg
\]

sample argument

\`
given

\text{op} \ \text{constructor} (u: \text{Var}, C: \text{Class}) \ // \ u = \text{new} C()
\text{post} \ \text{stack}' = \text{stack} + (u \rightarrow r) \ \text{and} \ \text{tag}' = \text{tag} + (r \rightarrow C) \ \text{for some fresh} \ r

\text{TYPECHECKS} \equiv u: cu \Rightarrow C \leq cu
\`

\`
we argue

by hypothesis, STACK-SAFE holds in prestate, so stack' is safe everywhere except u at u, we have tag'[\text{stack}'[u]] = C, and C \leq cu by TYPECHECKS
so STACK-SAFE holds in poststate
\`
preservation, another

another argument

• given

   \[
   \text{op fieldGet (u, v: Var, f: Field) // } u = v.f \\
   \text{pre } v \neq \text{null} \\
   \text{post stack'} = \text{stack ++ (u -> heap [stack[v]][f])}
   \]

   TYPECHECKS \equiv u: cu \land v: cv \land f: c \rightarrow cf \Rightarrow cv \leq c \land cf \leq cu

• we argue

   at u, tag'[stack'[u]] = tag[heap [stack[v]][f]]
   by HEAP-SAFE, tag[heap[stack[v]][f]] \leq cf, and cf \leq cu by TYPECHECKS
   so STACK-SAFE holds in poststate

note

HEAP-SAFE is trivially maintained for ops that only modify stack
type safety in practice
null dereferences

recall

- op had precondition to prevent null deref:

```plaintext
op fieldGet (u, v: Var, f: Field) // u = v.f
pre v != null
post stack' = stack ++ (u -> heap [stack[v]][f])
```

in practice

- statement throws exception
- variable is not bound
- control continues at point at which exception is caught
casts

recall

• statement fails to typecheck, but no runtime error
  
  List l = l2.elt; // OK, rejected

solution

• programmer inserts \texttt{cast}
  
  List l = (List) l2.elt;

• runtime checks that expression has type indicated by \texttt{cast}
• if not, throws \texttt{ClassCastException}
primitives

not all values are object references or null

• also have primitives (int, char, boolean)
• corresponding object types (Integer, Character, Boolean)
• ugly part of Java, but improved in Java 5 with automatic “un/boxing”
methods

method calls
• same issue as field access
• type checking ensures that methods exist
• same principles

semantically
• method call adds a new “frame” to the stack
• special variable “this” bound to receiver object
real proofs of type safety

real proofs

• use same method we used
• but fuller semantics
• and details checked mechanically

for example, see

Tobias Nipkow and David von Oheimb
Java Light is Typesafe -- Definitely
Proc. 25th ACM Symp. Principles of Programming Languages
http://david.von-oheimb.de/cs/papers/POPL98.html
safe languages

combination of
• type safety
• automatic memory management
• array bounds checking

ensures
• no illegal accesses
• program can only access memory location with a name

safe languages
• have these features
• support modular reasoning: unrelated module can’t trash your data
• are immune to buffer overflow attacks
review
summary

invariants
• powerful way to reason about state machines
• use induction on operations to prove property of all reachable states

state machine semantics
• called “small step semantics”
• many applications in design and theory of languages

type safety
• basis of safe languages